λ-IO: A Unified IO Stack for Computational Storage

Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu, *Tsinghua University*

https://www.usenix.org/conference/fast23/presentation/yang-zhe

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

λ-IO: A Unified IO Stack for Computational Storage

Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu*

Tsinghua University

Abstract

The emerging computational storage device offers an opportunity for in-storage computing. It alleviates the overhead of data movement between the host and the device, and thus accelerates data-intensive applications. In this paper, we present λ-IO, a unified IO stack managing both computation and storage resources across the host and the device. We propose a set of designs – interface, runtime, and scheduling – to tackle three critical issues. We implement λ-IO in full-stack software and hardware environment, and evaluate it with synthetic and real applications against Linux IO, showing up to 5.12× performance improvement.

1 Introduction

Data-intensive applications (e.g., data mining) suffer from the overhead of loading sizable data from the storage device for processing. Emerging computational storage devices offer the opportunity to alleviate the bother and thus gain increasing attention in recent years [1–3]. They encapsulate in-device computation resources, which enables the applications to offload the computation down into the devices for in-storage computing (ISC) [4, 5], therefore mitigating the overhead of data transfer between the host and the device.

As both the host and the device can conduct computation, we observe from experiments that blindly pushing computation tasks to the devices is not always optimal; instead, applications may be faster on either side regarding to various application features (e.g., computation complexity) and the ever-changing status of the host and the device (e.g., cache ratio, bandwidth consumption). Since the operating system has delivered a sophisticated IO stack with mature functionalities and interfaces, we explore a fundamental question in the paper – *how to build a unfied IO stack managing both computation and storage resources across the host and the device?* For answering the question, we first identify three critical issues to achieve the goal.

1) Interface. IO stack has been evolving the read and write interfaces [6, 7], which are widely used by existing applications. We are committed to endowing them with the computation capability of the host and the device at a minimum change, i.e., unified interfaces for both sides that are generic and rely on no specific file systems.

2) Runtime. IO stack contains components like the page cache, file system, and device driver for storage management, but has no computation runtime. Thus, we need to build a unified runtime that enables executing the same piece of computation task on either the host or the device. eBPF [8, 9] is the preference for constructing such runtime: it proposes a hardware-independent intermediate bytecode format, so that the program can be compiled once and run on various ISAs. However, eBPF is still inapplicable to ISC due to its overstrict static verification (§2.3).

3) Scheduling. IO stack features rich scheduling mechanisms for reads and writes, but computation makes the dispatching mechanism more complicated. As application performance varies over features and system status as mentioned above, we aim at detecting and dispatching requests to the faster side effectively and efficiently.

In this paper, we propose λ-IO, a unified IO stack that comes with three key designs to tackle the aforementioned issues. First, λ-IO extends IO stack with λ extension across the host and the device to support ISC. Besides normal IOs, the extended interfaces enable an application to submit λ requests to customize, load and invoke computational logic (λ function) during reading and writing a file. With the extended interfaces, developers can still use their familiar programming style to access and process file data, hiding the details of how computation tasks are executed and scheduled.

Second, λ-IO proposes unified λ runtime that crosses the host-and-device boundary with two counterparts. At the core of λ runtime is sBPF (s stands for storage). For managing and executing λ functions atop heterogeneous hardware of both sides, λ-IO extends eBPF to sBPF, which supports pointer access and dynamic-length loop (loops are bounded at runtime to a specified threshold). In contrast to static verification of

[*] Jiwu Shu is the corresponding author (shujw@tsinghua.edu.cn).

eBPF, sBPF brings in dynamic verification with extra information from λ-IO.

Third, λ-IO adopts dynamic request dispatching to designate requests to the faster side effectively and efficiently. For effectiveness, we model the execution time of a λ request in both sides, so that the λ-IO finds the faster side. For efficiency, λ-IO profiles partial requests periodically to determine variables in the execution time model.

We implement λ-IO in the full-stack software and hardware environment (§4). We modify Linux kernel to integrate λ extension and build an NVMe device with λ extension support on a real hardware platform. We compare λ-IO and vanilla Linux IO on synthetic applications, showing 5.12× improvement. We also port a real application, Spark SQL [10], to λ-IO and evaluate its end-to-end performance with TPC-H [11], showing up to 2.15× improvement. We open source λ-IO at `https://github.com/thustorage/lambda-io`.

In summary, we make the following contributions.

- We present λ-IO, a unified IO stack managing both computation and storage resources across the host and the device.
- We propose a set of designs, unified interfaces, λ runtime, and dynamic request dispatching, to exploit the benefits of both sides.
- We implement λ-IO in the full-stack software and hardware environment, and evaluate it with synthetic and real applications against vanilla Linux IO, showing significant performance improvement.

2 Background and Motivation

2.1 In-Storage Computing and IO Stack

In-storage computing (ISC) originates in the disk era [4, 12], aiming to ship computation closer to storage. It mitigates the data movement overhead and exploits the in-device internal bandwidth. ISC gets revitalized with the advent of SSDs [5, 13], as an SSD typically has more powerful computation resources and higher internal bandwidth. Besides the researches for specific domain acceleration [14–22], general frameworks [23–27] allow programmers to define and offload their own computational logic. They mostly focus on providing manipulation interfaces in the userspace and accelerating computation in the device, but do not fully exploit the host-side computation and storage resources.

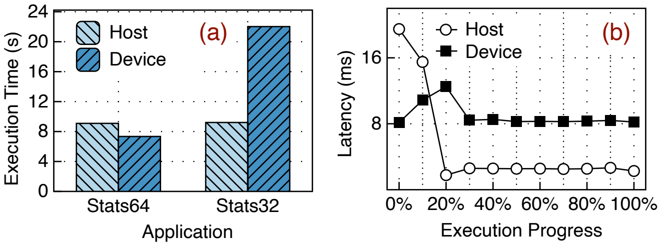

Figure 1: Performance Comparison of Host and Device. *Stats64 and Stats32 are two applications. They view the file data as 64-bit/32-bit integers and calculates the sum, maximum, and minimum. Refer to §5.2 for detailed settings.*

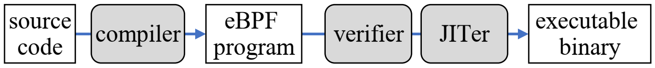

Figure 2: The Progress to Run eBPF/sBPF.

IO stack is a fundamental part of the operating system for managing storage devices, including the device driver, the block layer, and the file system. Although userspace IO libraries like SPDK [28] arise for their kernel-bypass low latency, the IO stack is still indispensable in most scenarios for three reasons. **1) Compatibility**. A sea of applications rely on POSIX file interfaces to access storage data. Programmers are also accustomed to leveraging APIs of the IO stack. **2) Functionality**. The IO stack offers abundant modules and functionalities, including the file system and the page cache. In contrast, a userspace IO library only supports data transfer with the raw storage device. The application has to build its own file system and data cache. **3) Sharing**. The IO stack has well-tested resource allocation and security mechanisms, so that users and applications can share the whole device. Sharing is also hard to be implemented in the userspace and absent from userspace IO libraries.

As ISC becomes increasing popular and the IO stack has unique advantages, we explore how to incorporate ISC to the IO stack in this paper.

2.2 Host-Device Coordination

We analyze the need for host-device coordination through application experiments. The key observation is that either the host or the device may be faster to run an application. 1) Different applications have different features and thus prefer different sides. We run Stats64 and Stats32 either in the host or the device (detailed settings in §5.2). As reported in Figure 1(a), Stats64 runs faster in the device while Stats32 runs faster in the host. 2) Even one application also favors different sides during the execution progress. We measure the request latency on both sides when running Stats64 with the warmed page cache (detailed settings in §5.4.1). As reported in Figure 1(b), earlier requests are faster in the device as the host does not cache data. The host outperforms significantly when it has the data in cache after 20%.

We conclude that blindly pushing down all the computational logic to the device may deliver suboptimal application performance. Many factors affect the execution time, such as computation complexity, data size, and cache. Thus, one should take factors into consideration and dispatch computation to their preferred side for better performance.

2.3 eBPF and its Limitations

eBPF (extended Berkeley Packet Filter) [8] is an in-kernel virtual machine. It enables the user to run a piece of logic inside the kernel without modifying the kernel source code or loading a kernel module [9]. Figure 2 shows the entire progress to run eBPF in three steps. 1) The user compiles the source code to an eBPF program in eBPF bytecode and loads the program via a specific syscall. 2) The in-kernel static verifier checks the program to ensure safety. 3) The just-in-time compiler (JITer) [8] translates the eBPF program to executable binary in native hardware for later execution. eBPF is used in a wide variety of domains such as network filtering, tracing, and profiling [9].

eBPF receives greater attention from in-storage computing researches recently [27, 29–34]. It becomes the preference for constructing the ISC runtime, because eBPF provides a hardware-independent bytecode format. The source code can be compiled to an eBPF program only once but runs on general CPUs (e.g. x86, ARM) and specific hardware (e.g. FPGA [35], ASIC [36]). It enables inserting user-defined logic to the devices [31, 35, 37] apart from the kernel.

However, we find that eBPF is inapplicable to general ISC because of its overstrict static verifier. eBPF lacks two critical features, pointer access and dynamic-length loop, which are widely employed in data processing code. We explain limitations with an example. `compute` function in Listing 1 is the typical computation code to sum values, but fails to pass the eBPF static verifier for two reasons. 1) Pointer access. The eBPF verifier checks that the program does not access arbitrary kernel addresses. As `input` and `output` are memory pointers, the verifier does not know their boundaries and prohibits pointer arithmetic and dereference. 2) Dynamic-length loop. eBPF checks each loop by simulating the iteration of the loop during the static verification. It supports a bounded loop [38] when the loop boundary is bounded so that the loop finishes in bounded time in the simulation. However, `length_i` is unknown during the static verification and is dynamically determined before execution. In the verifier's view, the loop is unbounded and the program does not complete in limited time. Therefore, it rejects the program.

We notice that XDP [37], a programmable network packet processing framework based on eBPF, proposed an explicit checking mechanism for pointer access. With the network packet content in the memory buffer [`data, data_end`), before dereferencing a pointer `data+offset`, the program is required to explicitly check `data+offset` < `data_end` in the code. However, XDP requires `offset` to be a known constant. So this explicit checking mechanism does not work for general ISC, where `offset` can be a variable.

In a nutshell, eBPF needs extension to embrace ISC without compromising safety.

3 Design

This section presents the design of λ-IO. We begin with an overview and describe a range of key techniques in detail.

Listing 1: Sample Pseudo-Code of Summing a File

```
// sum.c
ssize_t compute(void *output, void *input, size_t length_i) {
    int sum = 0;
    for (int i = 0; i < length_i / sizeof(int); i++) {
        sum += ((int *)input)[i];
    }
    *output = sum;
    return sizeof(int); // return the output size
}

// main.c
int vanilla_io_sum(int fd, int file_size) {
    char buf[BUF_SIZE];
    int sum = 0, sum_s;
    for (int i = 0; i < file_size; i += BUF_SIZE) {
        pread(fd, buf, BUF_SIZE, i);
        compute(&sum_s, buf, BUF_SIZE);
        sum += sum_s;
    }
    return sum;
}

int lambda_io_sum(int fd, int file_size) {
    int λ_id = load_λ("sum.sbpf");
    char buf[sizeof(int)];
    int sum = 0;
    for (int i = 0; i < file_size; i += BUF_SIZE) {
        pread_λ(fd, buf,  BUF_SIZE, i, λ_id);
        sum += *(int *)buf;
    }
    return sum;
}

int main() {
    int fd = open("path/to/file");
    vanilla_io_sum(fd, FILE_SIZE);
    lambda_io_sum(fd, FILE_SIZE);
    close(fd);
    return 0;
}
```

3.1 Overall Architecture

Figure 3 sketches the overall architecture of λ-IO. λ-IO extends the vanilla IO stack with λ extension to support offloading computation in the host kernel and the device. λ extension consists of three parts, λ-IO APIs, λ runtime (λ-kernel runtime and λ-device runtime), and request dispatcher.

λ-IO APIs (§3.2) introduce additional programming interfaces for applications. Besides normal IOs to a file, an application can submit λ requests to customize the computational logic (λ function), load the λ function to λ-IO, and invoke the λ function during reading and writing a file.

λ runtime (§3.3) crosses the host-and-device boundary with two counterparts, λ-kernel runtime and λ-device runtime. They provide identical interfaces of computation and data to execute a λ request. λ-IO extends eBPF to sBPF to support ISC, so that the λ function needs to be compiled and loaded only once, but runs in both sides.

Request dispatcher (§3.4) aims to designate λ requests to the λ-kernel runtime or the λ-device runtime effectively and efficiently. λ-IO profiles and monitors the status of the host and the device periodically. It estimates the request execution time of a λ request using our proposed model, and dispatches

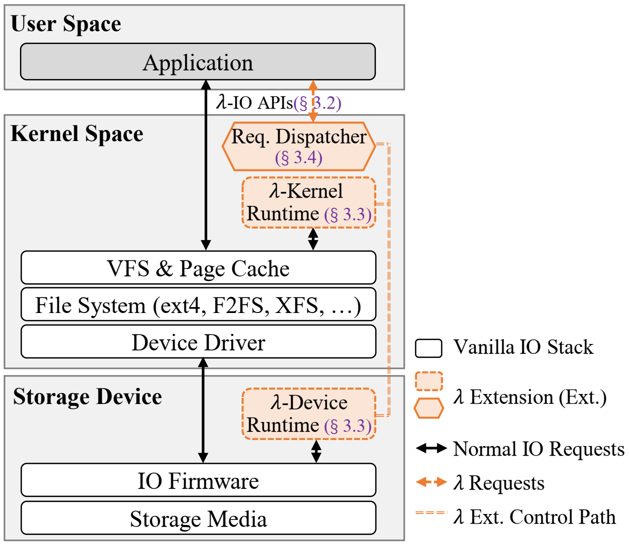

Figure 3: Overall Architecture of λ-IO.

Op	Interface
λ	`ssize_t compute(void *output, void *input, size_t length_i)`
load	`λ_id = load_λ(λ_path)`
open	`fd = open(file_path)`
close	`close(fd)`

read	`pread(fd, buf, length, offset)`
	`pread_λ(fd, buf, length, offset, λ_id)`

write	`pwrite(fd, buf, length, offset)`
	`pwrite_λ(fd, buf, length, offset, λ_id)`

* We omit some well-known types of parameters and return values.

Table 1: λ-IO APIs.

the request to the faster side.

3.2 λ-IO APIs and Workflow

λ-IO inherits the vanilla IO fundamental interfaces to open, close, read, and write a file, as listed in Table 1. Additionally, λ-IO introduces λ *load*, *read* and *write* interfaces, for an application to submit a λ request to offload computation during data transfer.

λ. The first line of Table 1 shows the interface to program the computational logic. Parameters of `input` and `output` point to input and output buffers, along with `length_i` to indicate the size of the input buffer. Thus, the computational logic in the function body can access data through pointers as it does in normal memory computing. It consumes data from the input buffer and produces data to the output buffer.

It is notable that the λ function body need not worry about the specific value of the two pointers, no matter the computational logic runs in the λ-kernel or the λ-device runtime. The λ runtime prepares memory buffers and sets proper values before executing the λ function.

Load. `load_λ` is the interface to load a λ function. The user compiles the λ function source code to an sBPF program, and loads it via `load_λ`. `load_λ` returns `λ_id` as the handle to be used in later λ read and write calls.

The user needs to compile and load a λ function only once, although λ-IO has two runtimes in the host kernel and the device. Receiving a loading call, λ-IO parses the sBPF program file, and transfers the bytecode to both the λ-kernel runtime and the λ-device runtime. Afterward, each runtime invokes the sBPF verifier and JITer to translate the bytecode to an executable binary in the native ISA for later execution.

Open and close. Two APIs remain intact. All λ extension APIs accept the identical `fd` as vanilla ones. Thus, the application can use normal and λ extension APIs simultaneously.

Read. Compared to normal read, λ read (`pread_λ`) adds a parameter `λ_id` to indicate the invoked λ funtion. λ-IO loads file data of the specified range as the `input`, performs computation of `λ_id`, and finally copies the `output` to the application-allocated buffer (`buf`).

To provide better intuition, we walk through how `pread_λ` works with an example of summing values in a file (Listing 1). `vanilla_io_sum` finishes the job via vanilla IO. The application repeatedly reads file data into a buffer (`buf`). Then it runs the computational logic (`compute`) to sum values. `lambda_io_sum` shows how to program the same logic using λ read (`pread_λ`). It has almost the same skeleton as `vanilla_io_sum`. The application loads the compiled sBPF program in Line 24, and gets a handle `λ_id`. When the application calls `pread_λ`, it additionally passes `λ_id`.

The λ runtime performs `pread_λ` in four steps. ❶ It loads the file data of the specified range into an input memory buffer, and sets the `input` and `length_i` parameters of the λ function. ❷ It allocates an output memory buffer and sets it as `output` of the λ function. The output memory buffer is as large as the input by default. ❸ It triggers the λ function, which sums data in the `input` and stores the result in the `output`. ❹ It copies data in `output` to the user-allocated `buf`, along with the output size represented by the return value of `compute`. In this way, the application only needs to allocate a buffer to receive the execution result, instead of the file data.

Write. λ write (`pwrite_λ`) works similarly to λ read in the reversed direction. λ-IO uses data in `buf` as the `input`, runs the function of `λ_id`, writes the `output` to the file at the `offset` and returns the output size.

The λ runtime performs `pwrite_λ` in four steps. ❶ It copies data of `buf` to a memory buffer in λ extension runtime, sets it as the `input` of λ. ❷ It allocates an output memory buffer and sets it as `output` of λ. ❸ It triggers the λ function, and stores the output data to the file from the position of `offset`. ❹ It returns the output size to the application.

3.3 Cross-Platform λ Runtime

The λ runtime plays a central role in executing λ requests of `load_λ`, `pread_λ`, and `pwrite_λ`. To execute λ requests in both the host kernel and the device, the λ runtime crosses the boundary of the host and the device with two instances, λ-kernel runtime and λ-device runtime.

To achieve the cross-platform runtime design, we identify two critical aspects of challenges, computation and data. For computation, both λ-kernel and λ-device runtimes have to store and run the same λ functions, although they are on top of different computation hardware platforms. For data, λ functions should be able to access consistent file data in λ-kernel and λ-device runtimes, along with other userspace applications. We describe how to tackle two challenges separately.

3.3.1 Computation: Extending eBPF to sBPF

At the core of the λ runtime is sBPF. As we state in §2.3, eBPF faces two critical limitations, pointer access and dynamic-length loop. We aim to tackle limitations efficiently. Our ***key idea*** is to bring in dynamic verifications with extra information from λ-IO, so as to check the pointer access and loop during running. sBPF inherits the bytecode format of eBPF, but extends the verifier and JITer.

For the pointer access, sBPF focuses on two memory buffer pointers, `input` and `output`. The sBPF verifier tracks all the pointer variables derived from `input` and `output`, by adding or subtracting an offset. For each dereference of such pointers (e.g. line 5 in Listing 1), the sBPF JITer inserts additional native code for checking the pointer during running. sBPF dynamically checks whether the dereferenced pointer falls within the given region $[\texttt{input}, \texttt{input} + \texttt{length\_i})$ or $[\texttt{output}, \texttt{output} + \texttt{length\_i})$, and returns an error when encountering an out-of-boundary pointer dereference. Afterward, the λ runtime terminates the execution of the `pread_λ` or `pwrite_λ` request with an error code to be handled by the application. In this way, sBPF ensures that pointer accesses to the input and output buffers are valid.

One may notice that `length_i` is passed by the application. For safety, λ-IO checks whether `length_i` is valid through kernel safety verifications (e.g. by `rw_verify_area`), before passing `length_i` to the λ function.

λ-IO does not introduce eBPF helper functions such as `bpf_probe_read` [39] to check pointer access. The root cause is that each eBPF helper function call is translated to a function call in the native binary by the JITer. If a program accesses `input` and `output` buffers via helper functions, the calls incur heavy overhead.

Note that sBPF only handles two pointers of `input` and `output` specially, as they are allocated and managed by λ runtime. Other pointers are still verified by the static verifier.

For the dynamic-length loop, sBPF applies a dynamic count. We observe that, the program has at least a jump-back instruction with a negative offset to implement a loop. Therefore, sBPF limits the number of executing jump-back instructions. The sBPF verifier allows loops during the static verification. The sBPF JITer allocates a counter and inserts extra native code beside each jump-back instruction. Once a jump-back instruction is executed, the counter increases. If the counter reaches the preset loop threshold, the program terminates and returns an error. As we limit the number of jump-backs, the program completes in bounded time.

We further describe how to choose the loop threshold value. For an ISC program, the number of loops is typically proportional to the input buffer size. In practice, the threshold can be set to the same order of magnitude as the maximum input buffer size of programs. In this way, λ-IO permits normal programs and aborts buggy or malicious programs.

Safety. sBPF changes the logic of pointer access and dynamic-length loop, but does not introduce extra safety risks compared to eBPF. In eBPF, the Linux kernel and the eBPF toolchain (e.g., verifier, and JITer) are trusted, while the user-written source code is untrusted. In sBPF, both the kernel and the eBPF toolchain are extended so that the source code running with sBPF can behave differently in two aspects:

1) Pointer accesses to the `input`/`output` buffer. The λ runtime allocates the input/output buffer and checks that the user has permission to access the file range (λ-IO strictly follows the ACL checking of the file system when accessing file data). Once a λ request finishes, the input/output buffer is reclaimed as well. Except for the input/output buffer, pointer accesses to other memory space are verified by eBPF's default static verifier. As a result, sBPF does not introduce extra risks of memory leak compared to eBPF.

2) Dynamic-length loops. sBPF does not allow infinite loops during execution; instead, sBPF allows loops to pass the verifier's checking, but lets the program abort if a loop repeats too many times.

3.3.2 Data: Consistent File Access

As we state in §3.2, the λ function accesses file data via the `input` pointer in `pread_λ` while via the `output` pointer in `pwrite_λ`. Given that there are a sea of file systems, such as ext4, F2FS and XFS, we introduce how λ-IO access file data consistently without relying on any specified file systems for compatibility. Our ***key idea*** is to leverage generic interfaces and components, e.g. existing syscalls, VFS, and page cache in the kernel, and to let the host control consistency.

λ-kernel runtime. Considering compatibility to a host of underlying file systems, we place the λ-kernel runtime atop VFS and page cache, as shown in Figure 3. Regardless of the specific file system, VFS provides the unified file abstraction and access interfaces. For a λ request, the λ-kernel runtime accesses file via `kernel_read` and `kernel_write`.

To optimize read performance, we propose kernel-version mmap, `kernel_mmap` to avoid the heavy data copy during `kernel_read`. λ-IO maps the pages located in the requested range to a kernel virtual address region via `vm_map_ram`. Afterward, λ-IO passes the mapped virtual address as the `input` pointer, so that the λ function can access the file directly.

λ-device runtime. Unlike the host end, the device is agnostic to the host-side file semantics. The device first has to know the exact storage locations of a range in the file. So the host is responsible for extracting necessary metadata and pushing it down to the device. Fortunately, Linux offers `FIEMAP` and `FIBMAP` ioctl interfaces [40, 41], to retrieve the file extent metadata of storage locations by given offset and length.

For λ read, λ-IO gets extent metadata of the specified range inside the file, and pushes the extent down to the device. The λ-device runtime loads the data from given storage locations to a continuous device memory buffer, and passes the buffer address to the λ function running in the device.

λ-IO does the similar for λ write inside the allocated scope of a file. In the case that a λ write appends the output file, λ-IO preallocates device storage space by `fallocate` [42] and then retrieves the file extent metadata of storage locations. Additionally, λ-IO resets the `unwritten` flag as previous works do [1, 25, 43]. In this way, the host can read data written by the λ write on the device, instead of zeros.

Consistency. As data may be modified in the host userspace, the host kernel, and the device, λ-IO has to guarantee data consistency among three places.

One is the host-side consistency between the userspace and the kernel. Both the userspace and the kernel rely on the identical VFS and page cache to access file data. As the λ-kernel runtime reads and writes file through `kernel_read` and `kernel_write`, these calls go through the same path as userspace calls. In this way, consistency is guaranteed by the VFS, page cache, and underlying file systems.

The other is the consistency between the host and the device. λ-IO maintains the consistency in the host. During executing a λ read/write request in the device, λ-IO acquires a read/write lock on the file, to guarantee consistency against other normal IO and λ requests. Additionally, λ-IO manipulates the host-side page cache before dispatching λ requests to the λ-device runtime. Before dispatching a λ read request to the device, λ-IO flushes dirty cache within the overlapped range, so that the device can view the up-to-date version. Before dispatching a λ write request to the device, λ-IO invalidates the host-side page cache within the overlapped range, so that the host can retrieve new data written on the device.

We propose λ-kernel not for performance gaining over vanilla IO, but for two strengths. 1) λ-kernel, together with λ-device, offers unified interfaces and runtime across the host and the device. On basis of this, the computational logic can be run and dispatched on both sides. 2) With the host-side components in the kernel, λ-IO can better exploit the capabilities of the vanilla Linux IO stack, e.g. compatibility, functionality, and sharing, as we mention in §2.1. In this way, λ-IO is friendly to programmers, since a sea of applications use POSIX file interfaces of the vanilla IO stack.

3.4 Dynamic Request Dispatching

λ-IO executes λ requests in both sides, but a request can be faster in either side, as we state in §2.2. The design goal of the request dispatcher is to designate λ requests to the faster side effectively and efficiently.

For effectiveness, we model the execution time of a λ request in both sides, so that the dispatcher can predict the execution time and find the faster side. For efficiency, the requester dispatcher should introduce low extra overhead to the execution. To this end, λ-IO periodically profiles partial requests to determine variables in the model, rather than profiles each λ request. We describe them in detail below.

3.4.1 Modeling Execution Time

We first consider λ read. We start by defining a few symbols, the loaded data size from the storage media D, the bandwidth between the storage media and the device controller buffer B_s for a request, the bandwidth between the device and the host B_d for a request, and the host-side computing bandwidth B_h for a request.

We notice that multiple requests may be submitted and executed concurrently. Therefore, these bandwidth variables correspond to a request, instead of all the concurrent requests. When multiple requests are executed concurrently, the data transfer and computation bandwidth of each request is affected by other requests and thus less than the aggregated.

The execution time in the host t_h is composed of transferring data from the storage media to the device controller, then to the host, and the host computing, as shown in Equation 1.

If the computation happens in the device, the transferred data size between the host and the device becomes αD, where α is the ratio between the output and input. The computing bandwidth in the device is βB_h, where β is the ratio between the device and the host computing bandwidth. We have the execution time t_d in Equation 1.

$$\begin{cases} t_h = \frac{D}{B_s} + \frac{D}{B_d} + \frac{D}{B_h} & if\ in\ the\ host \\ t_d = \frac{D}{B_s} + \frac{D}{\beta B_h} + \frac{\alpha D}{B_d} & if\ in\ the\ device \end{cases} \tag{1}$$

Similarly, given the input data size D, we have the following to calculate the execution time for a λ write

$$\begin{cases} t_h = \frac{D}{B_h} + \frac{\alpha D}{B_d} + \frac{\alpha D}{B_s} & if\ in\ the\ host \\ t_d = \frac{D}{B_d} + \frac{D}{\beta B_h} + \frac{\alpha D}{B_s} & if\ in\ the\ device \end{cases} \tag{2}$$

We further discuss the impact of the host-side cache on λ read. The cache reduces data transfer from the device and saves significant data movement overhead. Given the cache ratio c, Equation 1 becomes

$$\begin{cases} t_h = \frac{(1-c)D}{B_s} + \frac{(1-c)D}{B_d} + \frac{D}{B_h} & if\ in\ the\ host \\ t_d = \frac{D}{B_s} + \frac{\alpha D}{B_d} + \frac{D}{\beta B_h} & if\ in\ the\ device \end{cases} \tag{3}$$

The higher the ratio of the cached file data, the more we tend to dispatch the λ read request to the host.

3.4.2 Periodical Profiling and Dispatching

To estimate the execution time using the above equations, λ-IO has to determine variable values in the equations. We divide seven variables into two groups, one's exact values obtained directly and one estimated by profiling.

The first group includes the input data size D and the host-side cache ratio c. D is the `length_i` in Table 1. To get c, the request dispatcher walks the page cache tree to count the number of cached pages n_c within the given range. Afterward, it calculates c by $n_c \times PAGE\_SIZE /$ `length_i`.

The second group includes the remaining five variables, B_s, B_d, B_h, α, and β. We call them profiling variables. λ-IO does not know their exact values for a λ request in advance, so it estimates their values through periodical profiling. It is notable that B_s is the bandwidth for a request, but is not necessarily equal to the physical bandwidth between the storage media and the device controller buffer. B_s changes over time, like when multiple requests share the device, and so are the other variables. Thus we call these variables in the second group profiling variables and describe how to profile them.

We notice that profiling variables vary on different files and λ functions. Therefore, λ-IO profiles their values for each (`file_path`, `λ_id`) pair separately. The dispatching supports collocated applications as (`file_path`, `λ_id`) pairs may come from different applications running simultaneously.

To determine these variables efficiently, λ-IO profiles partial requests periodically rather than profiles each request. For a given (`file_path`, `λ_id`) pair, λ-IO sets a *profiling period*, like n requests. For the beginning k requests in a period (we refer to k as *profiling length*), λ-IO submits them to both the λ-kernel and λ-device runtimes. Each runtime measures the values of profiling variables of a request during execution. After k requests complete, λ-IO calculates the average of each variable and uses it as the estimated value.

For the following requests in the same period, λ-IO calculates the estimated execution time in both sides using the equations and estimated variable values. Then it dispatches the request to the faster side. When a new period comes, λ-IO repeats the profiling to estimate new values for the variables.

To achieve a balance between effectiveness and efficiency, λ-IO should be careful to set values of two profiling parameters, i.e. profiling period and profiling length. With a smaller profiling period and a bigger profiling length, the dispatcher can keep track of the status and find the faster side more effectively, but will induce higher overhead. In contrast, a bigger profiling period and a smaller profiling length lead to lower overhead, but the dispatcher may miss real-time changes in the status and make suboptimal decisions.

4 Implementation

We implement λ-IO in the full-stack software and hardware environment. We modify the Linux kernel to integrate λ extension in the host and build an NVMe device with λ extension support on a real hardware platform. We introduce the implementation details in this section.

Host. We implement the request dispatcher and the λ-kernel runtime in the host kernel with a kernel module. The kernel module creates a `procfs` [44] file to receive λ extension calls. We modify the eBPF verifier and the x86 JITer.

Device. We implement the device-side components on a real hardware platform, Daisy OpenSSD [45]. It is representative of computational storage devices, as one of the latest iterations of the OpenSSD project [46, 47]. OpenSSD is widely recognized and used by ISC researches from both industry and academia [18–21, 27, 48, 49]. The device connects to the host via PCIe and operates as an NVMe drive. We migrate OpenExpress [50, 51], an open-source NVMe controller, to the platform. We refactor the NVMe firmware thoroughly, modify the eBPF verifier and the ARM eBPF JITer from the kernel, and construct the λ-device runtime atop them.

Communication. We use standard NVMe over PCIe to communicate between the host and the device. λ-IO delivers normal IO requests through existing NVM IO commands, and customizes three *Vendor Specific Commands* [52, 53] for λ load, read, and write. λ-IO creates one standalone NVMe command for one λ request separately, delivering all necessary file extent metadata and indicating the λ function it triggers.

5 Evaluation

We evaluate λ-IO to answer the following questions:

- How does λ-IO perform on typical in-storage computing applications compared to existing approaches? (§5.2)
- How does λ-IO perform when collocated applications run concurrently? (§5.3)
- How do workload characteristics and dispatching configurations affect the performance? (§5.4)
- How is the sBPF overhead compared to eBPF? (§5.5)
- Can the end-to-end performance of a real application benefit from λ-IO? (§5.6)

5.1 Experimental Setup

Testbed. Table 2 shows our detailed testbed configurations. We implement λ-IO on the host and the device as stated in §4. The host CPU has 4 physical cores and 8 hyperthreads. The device SoC has 4 ARM cores. The 2GB SoC memory acts as the device controller memory. We equip the device with two 32GB DRAM DIMMs to act as the backend storage media, as the hardware board manufacturer provides no NAND flash modules. The data transfer bandwidth between the storage media and the device controller memory is 3.52GB/s. The bandwidth between the device controller memory and the host is 3.22GB/s. The bandwidth for ARM cores to access the device controller memory is 5.09GB/s. The performance is comparable to real SSDs and hardware platforms in recent works [19, 24, 48, 54].

Host	CPU	Intel Core i7-7700 @3.6GHz (4C8T)
	Memory	16GB
	OS	Ubuntu 20.04.2 LTS
	Kernel	Linux 5.10.21
Device	SoC	Xilinx Zynq Ultrascale+ ZU17EG
	Memory	2GB
	Storage	64GB

Table 2: Testbed Configurations.

App.	Description	ISC LoC	
		λ-IO	INSIDER [25]
Stats64	Read the file data as 64-bit integers and calculate the sum, maximum, and minimum.	30	240
Stats32	Read the file data as 32-bit integers and calculate the sum, maximum, and minimum.	30	240
KNN	Read vectors in the file and calculate distances between a target vector.	32	99
Grep	Read rows in the file and match a target string.	35	254
Bitmap	Decompress a bitmap and write to the file.	20	188

Table 3: Information of Synthetic Applications.

Workloads. We choose five synthetic applications Stats64, Stats32, KNN, Grep, and Bitmap with their information listed in Table 3. The first four applications read data from the source file and process it. The last application, Bitmap, decompresses a bitmap passed by `buf` and writes the output to the file. They fit the in-storage computing scenario well and are widely studied and evaluated in previous works [4, 13, 14, 24, 25, 27, 55]. The program code follows the skeleton of Listing 1 and costs little extra overhead to implement.

The input dataset file of four read applications is 16GB in size. The output file of Bitmap is also 16GB in size. As the output data also requires persistence, we synchronize the file after writing. We format the storage device to ext4 to store files. Applications run with 8 host threads and read/write file data to an 8MB buffer in each iteration. We set loop threshold (§3.3) to 16 million, profiling period (§3.4) n to 200, and profiling length k to 5. We keep the above configurations throughout experiments unless otherwise specified.

Porting overhead. As shown in Table 3, the LoC of implementing the same offloaded computational logic for λ-IO is significantly smaller than INSIDER [25]. We count the ISC LoC of INSIDER from its code repository [56]. The porting overhead of INSIDER is large, like other FPGA-based ISC frameworks. This is because INSIDER leverages HLS [57] to implement the computational logic on FPGA. Although HLS reduces the programming overhead, it still requires the programmer to be an expert in FPGA details [58] and write much hardware-specific code to optimize the performance.

Comparing targets. We adopt six IO modes in our evaluation, three vinalla IO modes and three λ-IO modes.

- ***(B)*** *Buffer IO*: uses the default IO mode of vanilla IO `pread`/`pwrite` to access file data. It enables the page cache. The computational logic is directly compiled to the native ISA, and so are Direct IO and Mmap.
- ***(I)*** *Direct IO*: similar to Buffer IO but opens the dataset file with `O_DIRECT`. It bypasses the kernel page cache.
- ***(M)*** *Mmap*: maps the dataset file to the userspace virtual address of the process. It eliminates data copy between the kernel and the userspace.
- ***(K)*** *λ-IO Kernel*: integrates the computational logic using λ-IO APIs and executes all the requests in the kernel runtime.
- ***(D)*** *λ-IO Device*: integrates the computational logic using λ-IO APIs and executes all the requests in the device runtime.
- ***(λ)*** *λ-IO*: integrates the computational logic using λ-IO APIs and leaves the λ-IO request scheduler to dispatch between the kernel and the device dynamically.

The first three IO modes (***B***, ***I***, ***M***) correspond to common approaches that use vanilla IO. (***D***) λ-IO Device corresponds to existing device-only ISC frameworks, e.g. Biscuit, INSIDER, MetalFS, BlockNDP [25–27, 55].

Warmup. We run experiments in two caching settings to examine the performance of λ-IO, 1) without warmup: drop the page cache before each execution, 2) with warmup: read the input/output file sequentially via Buffer IO, to warm up the page cache before each execution.

With warmup, we simulate the host-side cache state in practice and evaluate the performance of λ-IO in this scenario. After the warmup, partial data resides in the page cache while the rest is in the device if the input/output file is larger than the page cache. In real-time data analysis systems such as OLAP databases, the application continues producing data and storing it. Thus, the host-side cache always stores the latest part of the data. Meanwhile, the analytical application runs periodically to process the data in the recent period, where partial data resides in the cache while others have been flushed to the device.

5.2 Single Application

We compare the execution time of applications running singly in six IO modes. Figure 4 reports the results.

For further analysis, we break down execution time to three parts, IO, computation, and other. ❶ IO time means the time of reading/writing the file. In (B) Buffer IO and (I) Direct IO, it is the time of `pread`/`pwrite`. In (M) Mmap, the read time is included in computation, because the application retrieves data implicitly via page faults. In (K) λ-IO Kernel, it is the time of reading/writing data via the VFS. In (D) λ-IO Device, it is the time of transferring data between the storage media and the device controller memory. In (λ) λ-IO, it is the sum of IO time of requests, whether they execute in the host or the device. ❷ Computation time contains the calculation part of the application, along with memory access time. ❸ Other time is the remaining part excluding IO and computation.

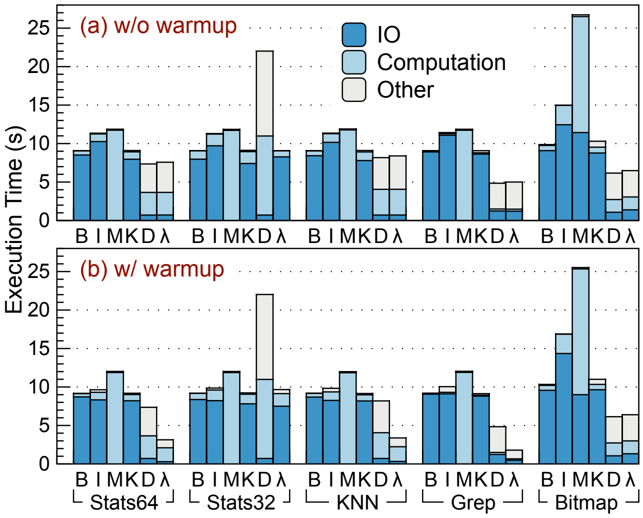

Figure 4: Performance of Applications Running Alone. ***B**: Buffer IO, **I**: Direct IO, **M**: Mmap, **K**: λ-IO Kernel, **D**: λ-IO Device, **λ**: λ-IO.*

1) λ-IO Device vs. vanilla IO. We choose Buffer IO to compare with λ-IO Device first, as it is the default operation mode of vanilla IO. We analyze the performance without warmup. In-storage computing improves the overall performance of Stats64, KNN, Grep, and Bitmap by 23.24%, 10.82%, 87.13%, and 60.15% against Buffer IO respectively. This is because applications running in the host are bounded by IO, which takes up more than 92.04% of execution time. For a normal read, data is transferred from the storage media to the device controller memory and then to the host software stack, as a normal SSD does. In contrast, IO time in λ-IO Device occupies less than 25.29%, because loading data inside the device avoids transferring to the host and going through the software stack. After the data is read to the device controller memory, device processors enjoy higher internal bandwidth to access data. Moreover, the computational logic outputs a smaller amount of data than the input file data, reducing the data transfer overhead to the host. Write (Bitmap) works similarly, except for the reversed direction.

The performance with warmup is close to that without warmup. This is because the host memory is 16GB in size, and the page cache capacity is just smaller than the dataset. After the warmup, all the input file is cached inside the page cache, except the beginning part. As the application runs again, it still accesses data from the very beginning. The beginning part misses in the page cache and evicts other pages, which leads to more and more cache misses. Finally, the page cache fails to buffer any data in fact.

One might note that *other* time of λ-IO Device is relatively large, as illustrated in Figure 4. This is because the host issues requests with 8 threads, while the device only has 4 threads to process. So a request has to pend in the queue and wait for the completion of previous requests. But request pending and waiting actually exist in all modes. For example in Buffer IO, in the progress of `pread`/`pwrite`, IO requests also pend and wait in queues throughout the IO stack. The overhead is included in the IO time.

λ-IO Device does not always outperform vanilla IO, e.g. on Stats32. λ-IO Device consumes 6.65× computation time against λ-IO Kernel on Stats32. We examine the generated native machine code of Stats32 and find that eBPF does not support 32-bit integers well. Specifically, eBPF programs use 64-bit registers even for 32-bit integers in Stats32. It introduces a pair of left and right shift instructions to clear the upper 32 bits of registers. This mechanism induces significant overhead in the device processors. We leave optimization of supporting 32-bit integers for future work.

2) λ-IO Kernel vs. vanilla IO. λ-IO Kernel executes the computational logic in the host kernel and exhibits similar performance against Buffer IO. We look deeper into the time breakdown of λ-IO Kernel. It takes slightly more time to compute, since sBPF incurs overhead compared to the native ISA. Oppositely, λ-IO Kernel reduces data copy from the kernel page cache to the userspace buffer. These two aspects together result in similar performance to Buffer IO.

We further examine three vanilla IO modes. As shown in Figure 4, Buffer IO performs the best. Direct IO is slower, as it bypasses the page cache and therefore misses the opportunity of readahead. We note that computation takes up almost all the execution time in Mmap on read applications. It is because the computational logic accepts the mmaped virtual memory address and loads data from the file system via user-agnostic page faults. Page fault handling in the kernel induces high overhead [59], so leads Mmap to the slowest mode. Different from read applications, Bitmap still spends time on IO, because it writes the mapped data via `msync`.

3) λ-IO modes. The rightmost bars in Figure 4 present the performance of λ-IO modes. Without warmup, the execution time of λ-IO is almost the minimum value of λ-IO Kernel and λ-IO Device on all applications, although the dispatcher introduces less than 4.98% overhead.

With warmup, λ-IO improves more significantly. Stats64, KNN, and Grep are 2.19×, 2.71×, 5.12× faster than Buffer IO. λ-IO is faster than both λ-IO Kernel and λ-IO Device. When the application accesses the beginning part of the file, the λ-IO request dispatcher finds that the device processes faster. It thus submits these requests to the device and keeps the page cache untouched. As the beginning part passes, the rest file data is buffered in the page cache. Then the dispatcher detects the kernel is better and submits requests. In this way, λ-IO takes full advantage of both the kernel and the device, achieving the best performance. The write application, Bitmap, does not gain extra benefits from warmup, compared to the performance without warmup. As we need to synchronize the data to the device after writing, it always finishes faster in the device side.

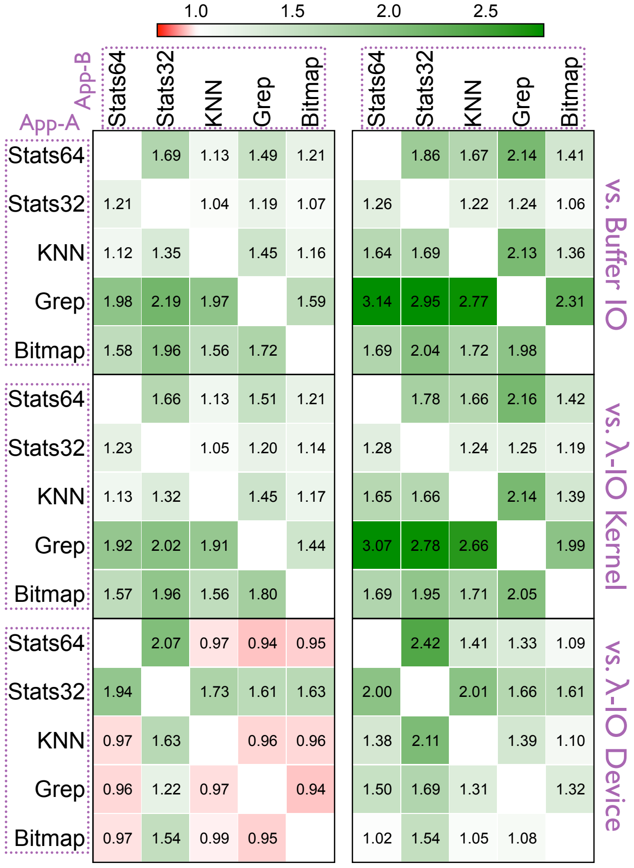

Figure 5: Speedup of Running Applications Concurrently in λ-IO Mode. *App-A and App-B are two collocated applications. Digits denote the speedup of App-A running in λ-IO against Buffer IO, λ-IO Kernel, and λ-IO Device.*

The results demonstrate that λ-IO dispatches requests effectively and efficiently. Moreover, λ-IO yields better performance than kernel-only and device-only ISC approaches for read applications running with warmup.

5.3 Collocated Applications

We evaluate executing collocated applications concurrently, to present how λ-IO works in this scenario.

With five applications, we have ten combinations of two different applications. For a combination, we run two applications concurrently, each with 4 threads and its own 16GB dataset. We evaluate the combination in Buffer IO, λ-IO Kernel, λ-IO Device, and λ-IO modes. Then we measure execution time of each application and calculate the speedup of each application in the λ-IO mode against other three modes. We draw results of all combinations in Figure 5, where digits mean the speedup of App-A.

For better understanding the figure, we take two digits 1.21 and 1.69 in the top left corner of subfigure (a) as an example. We run Stats32 and Stats64 concurrently in λ-IO mode. Compared to run them concurrently in Buffer IO mode, Stats32 and Stats64 complete 1.21× and 1.69× faster respectively.

As we evaluate in §5.2, Stats64, KNN, Grep, and Bitmap run faster in the device. We classify results of Figure 5(a) into two categories according to whether chosen applications both run faster in the device. 1) When Stats32 is chosen, λ-IO speeds up both applications by up to 2.19× compared to other three modes. This is because λ-IO dispatches Stats32 to the host and the other (Stats64, KNN, Grep, or Bitmap) to the device, so as to exploit both sides. 2) Otherwise when Stats32 is not chosen, λ-IO outperforms Buffer IO and λ-IO Kernel by up to 1.98×, because λ-IO executes them in the faster device side. Compared to λ-IO Device, λ-IO introduces less than 5.64% dispatching overhead. As shown in Figure 5.3(b), the speedup with warmup is higher than that without warmup, because the dispatcher is aware of the page cache.

Since collocated applications run different computational functions (λ) on different dataset files (`file_Path`), λ-IO estimates the execution time of their requests and dispatches separately (§3.4). Based on the design, λ-IO can dispatch collocated applications effectively and efficiently, as demonstrated in this experiment.

5.4 Sensitivity Analysis

5.4.1 Dataset Size

We evaluate the impacts of the dataset size on the performance with warmup. We take Stats64 as an example due to space limitation. Figure 6 reports the results in six modes with varying dataset sizes. We categorize them into three types, less than (8GB), approximately equal to (16GB), and greater than (24GB, 32GB, and 40GB) the page cache capacity.

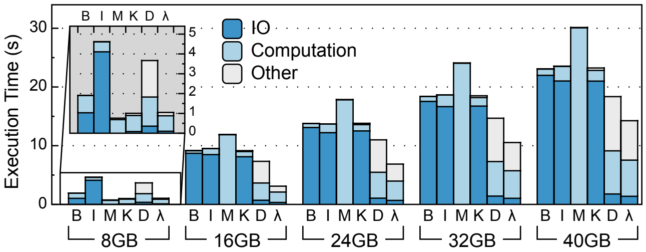

Figure 6: Stats64 Performance with Varying Dataset Sizes.

Dataset size < page cache capacity. Recall that the host memory is 16GB in our testbed, so the 8GB dataset fully resides in the page cache after the warmup run. The host does not have to read the data from the device, getting rid of the peripheral IO bottleneck. Direct IO still bypasses the page cache and loads data from the device, so it is significantly slower. The device has wimpier processors than the host, so λ-IO Device exhibits lower performance. λ-IO performance is close to λ-IO Kernel, which again proves the effectiveness of dynamic dispatching.

Dataset size ≈ page cache capacity. The page cache capacity is just narrower than the host memory 16GB size, as running

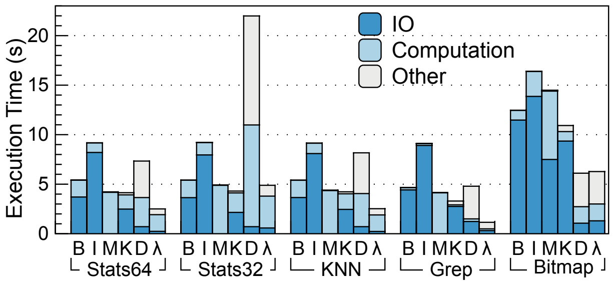

Figure 7: Performance with Random Warmup.

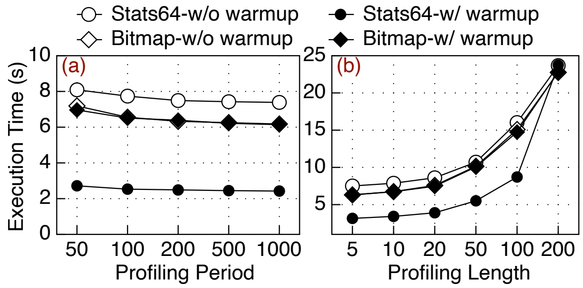

Figure 8: Performance with Varying Profiling Periods and Profiling Lengths.

programs occupy a fraction of space. This setting is the same as §5.2. The host-side page cache takes little effect and λ-IO exploits both the host kernel and the device simultaneously.

Dataset size > page cache capacity. When the dataset size exceeds page cache capacity, a fixed size of data is cached after the warmup. λ-IO performs better than the other five modes by 1.28× to 1.60× because it dispatches requests to both sides efficiently.

5.4.2 Warmup

We evaluate the impacts of the warmup approach on performance. As we state in §5.1, we warm up the page cache by sequentially reading the input/output file via Buffer IO. In this experiment, we change sequential read to random read and show the results in Figure 7.

Compared to results in Figure 4, Buffer IO performs better. After warmup by random read, random parts of the dataset file are in the page cache. During the execution, the application accesses the dataset file sequentially, so it can access partial data quickly when that part of the data is in the page cache. Nevertheless, λ-IO can still outperform Buffer IO by 4.05×. Dynamic request dispatching of λ-IO works effectively and efficiently, no matter how the warmup is taken.

5.4.3 Profiling Period and Profiling Length

Figure 8 depicts the performance with varying profiling periods and lengths. We choose Stats64 and Bitmap as the representative of read and write applications. When the profiling period becomes larger, the execution time decreases as λ-IO profiles fewer requests. The execution time remains stable when the profiling period is over 200, as the profiling and dispatching overhead is small enough. So we choose 200 as the default value of the profiling period because a small period leads to a quick response to the status.

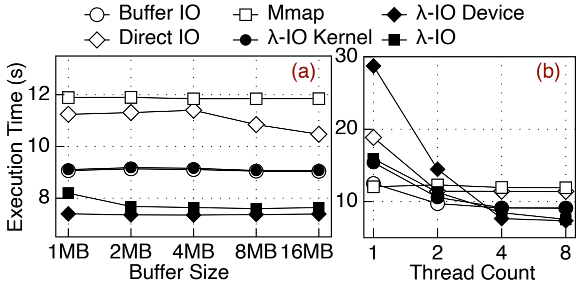

Figure 9: Stats64 Performance with Varying (a) Buffer Sizes and (b) Thread Counts.

When the profiling length goes up, the execution time increases significantly. With a profiling length of 200, λ-IO profiles all the requests and submits them to both the kernel and the device. We choose 5 as the default value of the profiling length. We calculate the average by removing the maximum and minimum values, to avoid accidental measurement errors.

5.4.4 Buffer Size

Figure 9(a) illustrates the performance of the Stats64 application with different data buffer sizes in each iteration. Most IO modes stay stable when the buffer size varies. Direct IO runs faster as the buffer size surpasses 4MB. Buffer IO and λ-IO Kernel perform almost the same, as in previous experiments.

5.4.5 Thread Count

Figure 9(b) plots the execution time of the Stats64 application with different number of host threads. Most IO modes execute faster when the number of threads grows up. Mmap rarely improves as using 1 thread reaches the top performance. The performance of λ-IO Device mode scales linearly from 1 thread to 4 threads. For all IO modes, 4 threads are enough to exploit the performance potential although the host CPU has 8 hyperthreads.

5.5 Overhead of sBPF

In this experiment, we evaluate the overhead of sBPF against eBPF. We choose two representative applications Stats64 and Stats32, because they run faster in the host and the device separately. We compare computation and total execution time in 8 settings, as shown in Table 4.

Settings. Eight settings are divided into two groups, kernel and device. We compile the computational logic to eBPF/sBPF programs. Two eBPF settings in the table mean running the bytecode by the eBPF verifier and JITer. We disable loop and pointer verifications to test the bare performance. DL checks dynamic-length loops on the basis of eBPF. DL+HF checks pointer access by helper functions on top of DL. We add two bpf helper functions for pointer read and write to the input and output buffers. All the input and output buffer

Setting		Stats64		Stats32	
		Compute (s)	Total (s)	Compute (s)	Total (s)
Kernel	eBPF	0.84	9.06	1.38	9.08
	DL	0.86	9.06	1.41	9.16
	DL+HF	3.99	9.47	9.19	14.15
	sBPF	0.99	9.06	1.60	9.12
Device	eBPF	2.39	6.38	8.64	18.75
	DL	2.64	6.84	9.04	19.54
	DL+HF	11.82	25.11	31.99	65.45
	sBPF	2.94	7.34	10.27	22.02

Table 4: Performance on eBPF and sBPF. *DL: eBPF with checking Dynamic-length Loops. DL+HF: eBPF with checking Dynamic-length Loop and checking memory access by Helper Functions.*

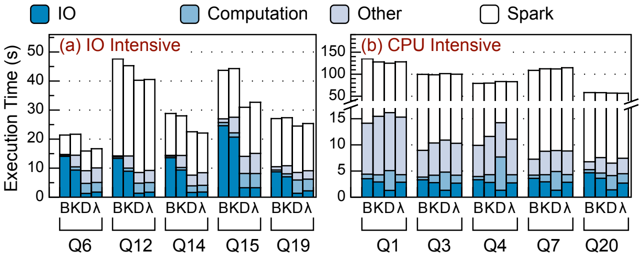

Figure 10: TPC-H Performance w/o Warmup. ***B**: Buffer IO, **K**: λ-IO Kernel, **D**: λ-IO Device, **λ**: λ-IO. **Q**`i`: query `i`.*

pointer accesses in the computational logic are modified to use helper functions. sBPF is the default setting used in all the other experiments. It combines dynamic loop and pointer access as we describe in §3.3.

Results. Table 4 depicts the results in 8 settings. Comparing the results of eBPF and sBPF settings, we find that the loop check induces at most 2.44% and 10.09% overhead in the host kernel and the device. Together with the pointer check, sBPF adds no more than 16.96% and 22.68% computation time in two sides respectively. The total execution time is almost unchanged in the host, and increases by 15.14% - 17.44% in the device. The results of UL+HF settings are notable. It expands computation time by up to 6.67× and 4.93× in two sides and slows down the total execution time seriously. The results demonstrate that loop and pointer checks of sBPF come at an acceptable cost.

5.6 Case Study: Spark SQL

In this experiment, we port a real application, Spark SQL [10], to λ-IO and evaluate its end-to-end performance with TPC-H [11]. Spark SQL is a prevalent SQL application for relational processing on structured data [60], as a module of Apache Spark. Spark SQL follows the schema of reading and processing data, so that it offers the opportunity of offloading computational logic to read via λ-IO. For an SQL query, Spark SQL first parses the query, constructs JAVA source code for processing data, and generates JAVA bytecode for later execution. Afterward, Spark SQL executes in two steps, 1) reads data from files, and 2) executes the JAVA bytecode to process input data and returns results. Thus, we can extract part of processing logic in the JAVA source code and integrate it into the first-step read via λ-IO.

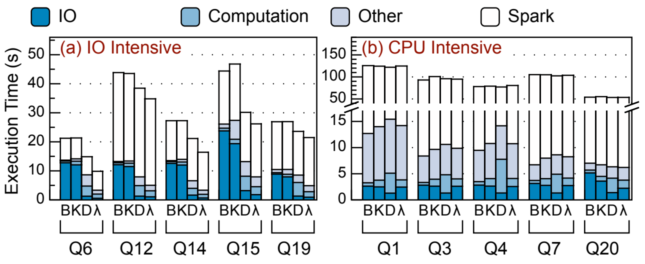

Figure 11: TPC-H Performance w/ Warmup.

Workloads. TPC-H is a widely-used OLAP benchmark [11] that defines a dataset of 8 tables and 22 SQL queries. We generate a 41.6GB dataset with a scale factor of 40. Three largest tables, `LINEITEM`, `ORDERS`, and `PARTSUPP` are 28.61GB (68.77%), 6.58GB (15.84%), and 4.53 GB (10.88%) in size respectively, together taking up 95.49% space of the whole dataset. We equip the host with 32GB of physical memory, which is still smaller than the dataset size. As for queries, we classify them into two categories, IO intensive and CPU intensive. We show 5 queries of each category due to the paper space limitation. They cover various SQL operations of projection, selection, aggregation, join, subquery, etc.

We focus on the largest table `LINEITEM` in this experiment, as all the 10 queries retrieve data from it. We add a preprocessing module between Spark SQL and λ-IO. For a SQL query, we extract filter logic (projection and selection) of `LINEITEM` from generated JAVA source code, convert it to C code, and integrate it to read in the preprocessing module. In this way, the execution progress finishes in two steps. 1) The preprocessing module reads file data and filters. 2) Spark SQL retrieves filtered data from the preprocessing module, further processes (e.g. aggregation, join), and returns results in the Spark environment. The filter logic of every query has less than 50 LoC and thus the porting overhead is low.

Comparing targets. We implement and run the preprocessing module in four modes, Buffer IO (B), λ-IO Kernel (K), λ-IO Device (D), and λ-IO (λ). In the Buffer IO mode, the preprocessing module reads file data via `pread` and runs the filter logic of projection and selection totally in the userspace. In the other three modes, the processing module integrates filter logic into read via `pread_λ` of λ-IO. We do not present the performance of the unmodified Spark SQL in the paper, as it is always slower than our modified Spark SQL with the preprocessing module, and instead, we use the Buffer IO mode as the baseline for fairness.

Results. Figure 10 and Figure 11 report experimental results without and with warmup. We warm up by reading the dataset sequentially before each execution, as we state in §5.1. As we mention in the settings above, the modified Spark SQL executes a SQL query in two steps, first in the preprocessing module and then in the Spark environment. We divide time in the preprocessing module into three parts of IO, computation, and other, as in Figure 4. Plus time in Spark, we break down end-to-end execution time into four parts in the figures.

Figure 10(a) shows the performance of IO intensive queries without warmup. In these queries, IO occupies 27.02% – 60.41% of end-to-end execution time in the Buffer IO mode. The execution time of λ-IO Kernel is similar to Buffer IO, like in previous experiments. λ-IO dispatches requests to the device efficiently and reduces preprocessing time (IO + computation + other) by up to 81.85% against Buffer IO. According to our statistics, the preprocessing module reduces the input data size to only 0.59% – 6.75%. Taking Spark time and scheduling overhead into account, λ-IO outperforms Buffer IO by 8.56% – 31.58%. λ-IO accelerates the end-to-end execution time of Spark SQL on IO intensive queries.

Figure 11(a) reports the performance with warmup. The dataset is larger than the page cache. We focus on the execution time of λ-IO. It performs faster than Buffer IO, λ-IO Kernel, and λ-IO Device by up to 2.15×, 2.16×, and 1.51× respectively. We also evaluate on a 20.8GB dataset (Scale Factor=20). With all data cached in the host, λ-IO Kernel is faster than λ-IO Device by 9.16% – 35.56%. λ-IO dispatches requests to the kernel with less than 2.98% overhead. These demonstrate the effectiveness of λ-IO dispatching.

Figure 10(b) and Figure 11(b) show performance of CPU intensive queries. λ-IO Kernel performs similarly to Buffer IO. Preprocessing time of Q20 in λ-IO Device is 16.28% less than λ-IO Kernel, where the row selectivity 15.1% is as low as IO-intensive queries. But for Q1, Q3, Q4 and Q7, preprocessing in the λ-IO Kernel is faster than λ-IO Device by up to 18.45%. This is because they select 30.3% – 98.5% rows, much higher than IO intensive queries. The device copies and returns more data and becomes less effecient than the host. Even though preprocessing is faster either in the host or the device, λ-IO chooses the faster side for all the 5 queries. The key difference from IO intensive queries is that Spark occupies dominant part, more than 87.48% of execution time. Therefore, the end-to-end execution time of all modes does not differ significantly.

In summary, taking Spark SQL as an example, real applications can benefit from λ-IO.

6 Related Work

In-storage computing (ISC) in the storage device originates in the disk era [4,12] and revives with the advent of SSDs [5,13]. We classify recent works into two categories, case study and general framework. Case study works accelerate a specific application or system by ISC, such as SQL [15,61,62], big data [14], graph [16,17,49], file system [22,29,63], and data training [18,19,64]. General ISC frameworks target offloading user-defined computational logic [23–27,55] and are closer to ours. As we state before, existing ISC frameworks mostly focus on providing manipulation interfaces in the userspace and accelerating computation in the device, but λ-IO redesigns the IO stack to support offloading computation. We discuss two works in more detail. Summarizer [24] proposes an automatic dispatching approach to saturate the device first, but does not consider many factors affecting the execution time in both sides. λ-IO takes many factors into consideration, and proposes profiling-based dynamic dispatching to designate requests. MetalFS [26] integrates into Linux as a file system driver, it only offloads computation to the FPGA device, without utilizing the host computation resources. λ-IO employs dynamic dispatching to exploit both sides.

As the network system continues delving into eBPF [35, 37], it also gains increasing attention in the storage system [29], especially ISC researches [31–34]. ExtFuse [29] accelerates file systems by embedding specialized request handlers into the kernel. Kourtis et al. [31] propose pushing computation to the disaggregated storage device, in order to avoid multiple network roundtrips. As the first one targets a specific acceleration, other ISC works make preliminary exploration and envision of bringing eBPF to in-storage computing. Zhong et al. [30,65] focus on pointer-chasing storage functions, e.g. a chain of IO requests, and aim to resubmit them in a lower layer of the host kernel to alleviate software stack overhead. But they do not pay enough attention to data computation inside one IO or offloading computation to computational storage devices. λ-IO analyzes and identifies two critical limitations that render eBPF inapplicable to general ISC. Responding to this issue, λ-IO proposes sBPF to break the limitations. Moreover, we build λ-IO on the full-stack software and hardware environment, not only the host side.

7 Conclusion

In this paper, we present λ-IO, which extends Linux IO to enable offloading computation to both the host kernel and the device. It carries and executes a user-defined computational logic during data transfer. We implement λ-IO in the full-stack and real software and hardware environment, and evaluate it with synthetic and real applications against vanilla Linux IO, showing significant performance improvement.

Acknowledgments

We sincerely thank our shepherd Alex Conway for helping us improve the paper. We also thank the anonymous reviewers for their feedback. This work is supported by the National Key R&D Program of China (Grant No. 2021YFB0300500), the National Natural Science Foundation of China (Grant No. 61832011, 62022051, & 62202255), and Huawei.

References

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

[2] SmartSSD - Samsung Semiconductor. https://samsungsemiconductor-us.com/smartssd/.

[3] Computaional Storage | SNIA. https://www.snia.org/computational.

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[5] 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, 2013.

[6] pread(2) — Linux manual page. https://man7.org/linux/man-pages/man2/pread.2.html.

[7] readv(2) — Linux manual page. https://man7.org/linux/man-pages/man2/readv.2.html.

[8] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[9] eBPF - Introdcution, Tutorials & Community Resources. https://ebpf.io/.

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1383–1394, New York, NY, USA, 2015. Association for Computing Machinery.

[11] TPC-H Homepage. https://www.tpc.org/tpch/.

[12] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. In *Proceedings of 24th Conference on Very Large Databases*, pages 62–73. Citeseer, 1998.

[13] 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*, pages 91–102, 2013.

[14] 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. IEEE, 2013.

[15] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[16] 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. *Proceedings of the VLDB Endowment*, 10(12):1706–1717, 2017.

[17] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. Grafboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 411–424. IEEE Press, 2018.

[18] 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 (USENIXATC 19)*, pages 395–410, 2019.

[19] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. 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*, ASPLOS 2021, page 717–729, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Xiaohao Wang, Yifan Yuan, You Zhou, Chance C. Coats, and Jian Huang. Project almanac: A time-traveling solid-state drive. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] 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.

[22] 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.

[23] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 67–80, USA, 2014. USENIX Association.

[24] 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.

[25] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing in-storage computing system for emerging high-performance drive. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 379–394, 2019.

[26] 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.

[27] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. Blockndp: Block-storage near data processing. In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, page 8–15, New York, NY, USA, 2020. Association for Computing Machinery.

[28] 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. IEEE, 2017.

[29] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIXATC 19)*, pages 121–134, 2019.

[30] 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*, pages 128–135, 2021.

[31] Kornilios Kourtis, Animesh Trivedi, and Nikolas Ioannou. Safe and efficient remote application code execution on disaggregated nvm storage with ebpf. *arXiv preprint arXiv:2002.11528*, 2020.

[32] Wenjun Huang and Marcus Paradies. An evaluation of webassembly and ebpf as offloading mechanisms in the context of computational storage. *CoRR*, abs/2111.01947, 2021.

[33] Giulia Frascaria, Animesh Trivedi, and Lin Wang. A case for a programmable edge storage middleware. *CoRR*, abs/2111.14720, 2021.

[34] Corne Lukken, Giulia Frascaria, and Animesh Trivedi. ZCSD: a computational storage device over zoned namespaces (ZNS) ssds. *CoRR*, abs/2112.00142, 2021.

[35] 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. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[36] eBPF Introduction - Netronome. `https://www.netronome.com/technology/ebpf/`.

[37] 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*, pages 54–66, 2018.

[38] Bounded loops in BPF for the 5.3 kernel. `https://lwn.net/Articles/794934/`.

[39] bpf-helpers Linux manual page. `https://man7.org/linux/man-pages/man7/bpf-helpers.7.html`.

[40] filefrag(8) - linux manual page. `https://man7.org/linux/man-pages/man8/filefrag.8.html`.

[41] Fiemap ioctl. `https://www.kernel.org/doc/Documentation/filesystems/fiemap.txt`.

[42] fallocate(2) - linux manual page. `https://man7.org/linux/man-pages/man2/fallocate.2.html`.

[43] Ian F. Adams, John Keys, and Michael P. Mesnier. Respecting the block interface - computational storage using virtual objects. In *Proceedings of the 11th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'19, page 10, USA, 2019. USENIX Association.

[44] proc(5) - Linux manual page. https://man7.org/linux/man-pages/man5/proc.5.html.

[45] Daisy OpenSSD Platform. https://www.crz-tech.com/crz/article/daisy/.

[46] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Trans. Storage*, 16(3), jul 2020.

[47] CRZ-Technology OpenSSD. https://github.com/CRZ-Technology/OpenSSD-OpenChannelSSD.

[48] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 147–162, 2021.

[49] 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*, pages 116–128, 2019.

[50] Myoungsoo Jung. Openexpress: Fully hardware automated open research framework for future fast nvme devices. In *2020 USENIX Annual Technical Conference (USENIXATC 20)*, pages 649–656, 2020.

[51] OpenExpresss Download Page. https://openexpress.camelab.org.

[52] NVM Express Base Specification 2.0. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-5.pdf.

[53] NVM Express NVM Command Set Specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-2021.06.02-Ratified-1.pdf.

[54] Intel Optane SSD 9 Series. https://www.intel.com/content/www/us/en/products/details/memory-storage/consumer-ssds/optane-ssd-9-series.html.

[55] 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 *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, 2016.

[56] zainryan/INSIDER-System: An FPGA-based full-stack in-storage computing system. https://github.com/zainryan/INSIDER-System.

[57] HLS Pragmas. https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html.

[58] Vivado 2021.2 - High-Level Synthesis (C based). https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0012-vivado-high-level-synthesis-hub.html.

[59] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX-ATC 20)*, pages 813–827, 2020.

[60] Spark SQL and DataFrames | Apache Spark. https://spark.apache.org/sql/.

[61] 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*, pages 1221–1230, 2013.

[62] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, 2014.

[63] Youyou Lu, Jiwu Shu, and Weimin Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, page 257–270, USA, 2013. USENIX Association.

[64] 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, 2021.

[65] 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.