Flexible NVMe Request Routing for Virtual Machines

Tu Dinh Ngoc, Boris Teabe, Georges Da Costa, Daniel Hagimont
IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, Toulouse, France

***Abstract*—Recent advances in storage hardware have resulted in massive improvements in both I/O latency and throughput. However, existing storage virtualization tools either depend on a heavy and inefficient I/O stack that is not optimized for parallelism, or require a separate API that is difficult to manage and monitor. In this work, we introduce NVMetro, a solution based on the NVMe protocol that proposes a flexible choice between multiple I/O paths to ease the development of adaptive and performant virtual storage. NVMetro provides two components: (1) an intelligent I/O classification and routing framework powered by eBPF; and (2) an easy-to-use and performant API to assist the creation of userspace I/O functions within our framework. We demonstrate the benefits of NVMetro by implementing two virtual storage functions, and we evaluate them using various benchmarks. The obtained results show that NVMetro achieves a performance and scalability comparable to bleeding-edge, kernel-bypass technologies while retaining the flexibility of traditional OS-based storage APIs.**

I. Introduction

The Non-Volatile Memory Express (NVMe) specification [1] has been widely adopted as a way to remedy I/O inefficiencies between the storage device and operating system. At the core of NVMe is the concept of I/O queues, where multiple independent storage operations can be performed simultaneously without the cost of synchronization. This highly-optimized design has massively benefited storage device and application scalability. Indeed, NVMe devices have managed to reach impressive performance figures; for example, the Intel Optane P5800X series claims a performance of up to 5 million I/O per second and a 99th percentile latency of less than 6 μs [2].

The increasing use of virtualization for managing and isolating system partitions is followed with increasing demand for efficient storage virtualization, in the form of *storage functions* covering certain use cases, e.g. data encryption, compression, replication, etc. Most of these technologies take one of two forms: a hypervisor-based, fully-featured stack that makes extensive use of OS features (e.g. QEMU's own virtual disk implementation, Linux's in-kernel Vhost), or a hardware-based stack that forgoes OS-level management in return for improved performance (e.g. device passthrough, SPDK [3]).

Hardware-based stacks deliver the most storage performance to VMs, but have the drawbacks of reduced manageability, difficulty of use and a limited feature set. For instance, device passthrough-based frameworks like SPDK assign an entire device to its userspace driver. As a result, that device cannot be accessed via the kernel API; disk access must be done with SPDK-specific APIs, or through a compatibility bridge (e.g. the DPDK kernel-native interface [4] for networking or FUSE [5] for virtual filesystems). Userspace solutions like FUSE also come with issues that can severely degrade performance [6]. Alternatively, single-root I/O virtualization (SR-IOV) can be used to partition one physical device into *PCIe virtual functions* that can be independently shared to each VM; however, SR-IOV is restricted to a single use case of inter-VM isolation, and gives the host little control or visibility over how each VM uses its resource partition, as the guest NVMe driver directly communicates with hardware, bypassing the host hypervisor.

All in all, these drawbacks are reasons to choose in-kernel I/O implementations over a higher-performing userspace or hardware-based one [7]. Yet OS-based stacks, while being easier to use, struggle to keep up with hardware. At the level of I/O performance demonstrated by the Intel Optane P5800X cited above, software becomes a significant part of I/O overhead, with kernel code taking nearly half of the time cost of a read/write system call [8]. Moreover, the implementation of complex storage functions can be challenging due to a lack of tooling integration in the kernel. Consider an example of a storage function that performs data encryption using Intel SGX. Such an application can be easily written with Intel's existing SDK; however, implementing said function inside the kernel requires a new, kernel-specific SDK, which is a considerably more challenging task. To summarize, current solutions lack the flexibility for implementing complex storage software; therefore, *we would need a more scalable and adaptable solution that meets all the challenges of virtual storage.*

In this work, we present *NVMetro*, a solution for efficiently managing storage in virtual machines. With NVMetro, we aim to ease the creation of flexible storage logic *without sacrificing either strong performance or security*. NVMetro proposes an unique solution that offers multiple I/O paths for handling virtual storage requests: (1) a *fast path* adjacent to hardware NVMe devices; (2) a *kernel path* attached to the host's kernel storage stack; and finally (3) a *notify path* controlled by an userspace I/O function. These paths are controlled by a central *I/O router* with interfaces for specifying policies to choose a best path for each individual request, as well as a support framework for userspace applications using the notify path.

Our design of NVMetro is guided by five main criteria:

- **Flexibility**: the **key ability** of NVMetro to provide fine-grained partitioning and control of storage requests, thus letting it adapt to multiple types of storage functions;
- **Performance**: that NVMetro does not significantly degrade performance compared to other solutions;

- **Isolation**: making sure that NVMetro does not break the security model of storage virtualization;
- **Compatibility**: ensuring that NVMetro works with all VMs supporting the NVMe specification;
- **Ease of use**: creation of a storage framework that minimizes the development effort needed to write a storage function with NVMetro compared to existing solutions.

To accomplish our design criteria, NVMetro uses two main components: (1) an *I/O router* supporting *pluggable classifiers* based on Linux's *Extended Berkeley Packet Filter* (eBPF) for encoding custom logic into the storage virtualization pipeline; and (2) a kernel-user API that assists the creation of *userspace I/O functions* (UIFs) for high-performance storage processing.

In Section II, we go into a deeper review of the NVMe and eBPF technologies to illustrate their relevance to storage virtualization. In Section III, we explain the goals and design criteria of NVMetro, show their applicability to a range of storage function use cases, and make comparisons to the designs of other works. Following these criteria, we present the main design of NVMetro's I/O routing, classification and UIF components. Section IV investigates two NVMetro use cases in depth: a data encryption function and a data replication function. We show in Section V the performance of NVMetro and its use cases under various workloads. Our results demonstrate that NVMetro takes good advantage of the storage performance of modern hardware, with our NVMetro-based disk encryption for VMs being up to $3.7\times$ faster than an in-kernel virtual encrypted disk based on `dm-crypt+vhost-scsi` while using as little as $0.9\times$ the CPU during heavy loads. Section VI gives an overview of other storage virtualization and computational storage approaches, and Section VII concludes our article.

II. Background

In this section, we present a background of technologies that make up NVMetro to help understand our solution.

A. NVM Express specification

The NVM Express specification defines a communication protocol between software (the "host") and storage devices ("controllers"). It specifies an *admin command set* for the host to interrogate and manipulate the controller, and various other command sets for individual use cases: the *NVM command set* for traditional block devices; and the *KV command set* for devices having a key-value interface.

NVMe provides a generalized *command queue* abstraction regardless of command set. The host sends I/O commands to a controller via *submission queues* (SQs); the controller processes them and puts their results into a corresponding *completion queue* (CQ). Aside from a dedicated SQ/CQ pair for admin commands, each NVMe controller can communicate with the host using up to 64K queues, each capable of holding up to 64K commands being processed in parallel. Each queue is a lockless producer-consumer ring buffer; as such, each CPU communicates with the controller using a dedicated queue, removing the need for synchronization when submitting requests. In addition, NVMe allows a N-to-1 correlation between SQs and CQs; in other words, multiple SQs can be associated to the same CQ. The host waits for completion notifications from a controller in two ways: it can either receive interrupts from the controller, or continuously poll its CQs for any new entries (called *busy polling*, a.k.a. *active polling*).

NVMe specifies various transports for moving I/O data, such as a *memory transport* for devices attached to a system bus like PCI Express, *message transport* over TCP or Fibre Channel, or a *RDMA-based transport* for high-speed remote storage over Infiniband or converged Ethernet. These transports let operating systems and applications use the same NVMe driver and software stack regardless of the underlying connection.

In summary, NVMe's scalable protocol and feature set enables countless new use cases: remote storage, intelligent tiering, key-value databases, etc. NVMe enjoys widespread support from numerous hardware and software vendors, and is poised to become a prominent all-purpose storage protocol.

B. eBPF virtual machine

Berkeley Packet Filters (BPF) [9] were introduced in the BSD operating system for packet inspection, filtering and capturing. BPF makes use of *BPF filters* written in an interpreted language, where a filter processes multiple protocols at different network layers. Linux originally adopted BPF in its socket filter [10].

The BPF instruction set was extended in Linux into *Extended BPF* (eBPF) with extra instructions and registers. Before running each eBPF program, the Linux kernel verifies its safety through a large range of properties, including constraints on memory accesses, loops and program size. eBPF programs can call a list of authorized kernel helper functions; however, this approach requires recompiling and reinitializing the eBPF verifier every time a new helper function is needed. Linux eBPF is currently employed in various use cases, such as system call filtering (via the Seccomp-BPF API), kernel tracing, LSM security controls, or infrared signal decoding. Notably, its Express Data Path (XDP) feature executes eBPF programs at the earliest points of network packet reception, such as in the network driver or directly inside SmartNIC hardware for the purposes of packet classification and routing [11].

III. Design of NVMetro

In the following sections, we give a general overview of our solution, then describe each design criterion in further depth.

A. General overview

NVMetro aims to ease the development of fast and flexible storage functions for VMs. We continue from our observation that current storage virtualization solutions only provide one possible access method; they are "all-or-nothing" in that once a storage function selects a specific virtualization API, it cannot easily switch to another API to meet new requirements. In NVMetro, we give developers multiple ways to process I/O requests with various trade-offs between performance, flexibility and ease-of-use depending on their use case.

Our solution operates in the hypervisor, and presents itself as a virtual NVMe controller in each concerned VM,

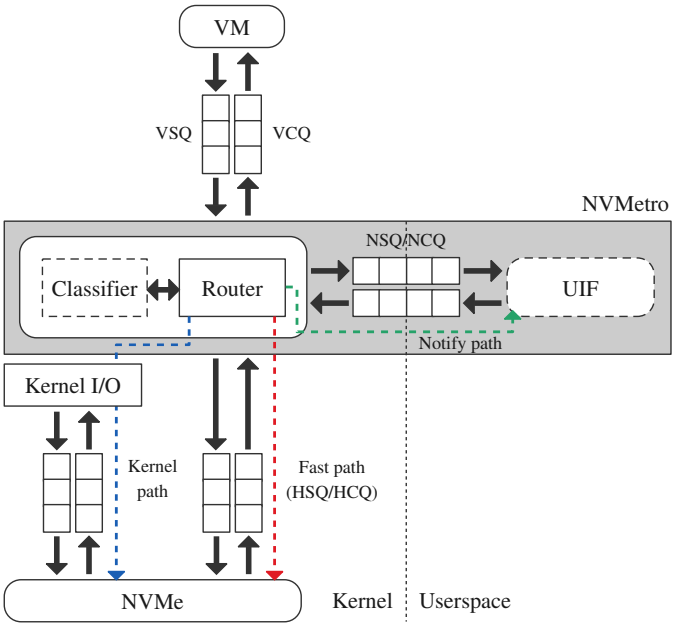

Fig. 1: NVMetro architecture and I/O paths. Customizable components are drawn in dashed outline. VSQ/VCQ, HSQ/HCQ and NSQ/NCQ denote *virtual*, *host* and *notify* submission and completion queues (see Section III-C).

intercepting and servicing I/O requests from the VM. This is done in accordance with the NVMe protocol, i.e. all VMs supporting NVMe work with NVMetro by default without guest modifications. Virtual controllers can be attached to an entire NVMe namespace on the drive, or a fixed partition of that namespace. NVMetro also supports creating multiple queue pairs, preserving NVMe's parallelism benefits.

Figure 1 summarizes NVMetro's main components. In short, requests from the VM pass through an *I/O router*. This router is modified by a customizable *I/O classifier* to route requests through one of three *I/O paths*: (1) a *fast path* to a physical NVMe device (red arrow); (2) a *kernel path* (in blue); and (3) a *notify path* (in green) to an external UIF.

The I/O router inspects incoming requests to find the most appropriate I/O path. As the path selection must depend on each particular function's use case, this step is done through *I/O classifiers* provided by the storage function. To determine a request's I/O path as quickly as possible, classifiers run directly inside the host kernel inside an isolated environment.

Next, we describe the tradeoffs to each I/O path that the classifier must take into account. The fast path, our simplest I/O path, involves sending each request directly to an underlying NVMe device. As a result, it is the most performant I/O path in NVMetro for most requests. The kernel path translates requests and sends them through the host kernel's block device architecture. This path incurs a request translation cost, and is only usable with requests that follow the Linux kernel's storage semantics (versus NVMe-specific or vendor-specific commands); however, it is compatible with Linux's block layer features (e.g. device mapper), as well as non-NVMe backends.

NVMetro's *notify path* exports requests for processing outside of the host kernel. Said requests are handled by an *userspace I/O function* (UIF) making up part of the desired storage function. We suggest using UIFs when either a) in-kernel request processing architecture is insufficient; or b) extra isolation of the storage function is required. To ease the creation of UIFs, NVMetro includes a C++-based framework that takes care of basic UIF housekeeping tasks.

Unique to NVMetro is that our I/O classifiers can run multiple times for each request. Each execution of a classifier dictates the next destination to which it should be sent. This feature assists complex use cases where a request needs multiple processing stages before it could be completed, by forming a state machine where the classifier models each request's transition between several states until it is completely fulfilled. Rather than filtering at every level of the I/O stack, the classifier is only invoked at key decision points during its lifetime, thus saving CPU and memory usage.

B. NVMetro's design criteria in detail

Flexibility. In existing storage stacks (MDev-NVMe [12], Linux's device mapper, SPDK, etc.), functionalities such as encryption, quality-of-service, etc. need to be implemented in the stack itself. In contrast, NVMetro provides a more fine-grained storage filtering, where UIFs are integrated into the I/O request path only as needed. In complex use cases such as encrypted key-value stores, NVMetro eases the integration of relevant technologies (e.g. Intel SGX) without affecting unrelated requests thanks to our router and classifier architecture.

In NVMetro, UIFs can be combined generically, by programming the I/O classifier to forward requests between UIFs; by direct IPC between UIFs; or by combining all function logics into a single UIF. Moreover, our modular design lets storage administrators install, migrate and remove storage functions on the fly, a desirable feature for avoiding VM reboots needed in some solutions (e.g. `vhost-scsi`).

Performance. NVMetro adds routing on top of the MDev-NVMe storage virtualization system; therefore, it necessarily imposes some overhead over MDev-NVMe. However, our key contribution is shortcut processing of I/O requests using a custom classifier followed by redirecting to the next hop as quickly as possible. In other words, commands that can be served directly by the physical disk are immediately sent for processing, and only those requiring extra processing will be sent to a slower I/O path. NVMetro thus maintains the benefit of I/O mediation without significantly impacting performance.

Isolation. In-kernel storage virtualization (Vhost, MDev-NVMe) keeps functional logic (e.g. encryption, replication, caching...) inside the kernel for performance reasons. This decision also increases the attack surface of these solutions. In NVMetro, we offload most of the work to isolated subsystems such as sandboxed classifiers and userspace processes. The remaining I/O router components in the kernel only minimally processes incoming requests, thus reducing our attack surface.

Compatibility. The large software stack in e.g. Vhost complexifies the implementation of certain I/O commands (e.g. block unmapping, security commands). Any storage layer lacking support for such commands will prevent them from being used by the guest. In contrast, besides NVMetro's support for the base NVMe specs, commands authorized by the I/O classifier can be passed directly to hardware, enabling the use of vendor extensions for performance or security purposes. NVMetro also easily adapts to new NVMe features (e.g. the KV command set) by changing the classifier without affecting the host kernel.

Ease of use. Kernel-bypass solutions like SPDK and `vfio-user` provide high performance through userspace polling drivers. However, these drivers require significant reengineering, take up exclusive control over the device, and cannot use features already built into the Linux kernel. This is, among other reasons, why Cloudflare chose to use the Linux kernel's TCP networking stack rather than DPDK [7]. In NVMetro, each UIF chooses the programming languages, libraries and APIs that best suit its purposes. We demonstrate this property by implementing an UIF based on the Intel SGX SDK.

Following the design criteria discussed above, the next sections detail NVMetro's two core components: the *I/O router and classifier*, and its accompanying *userspace I/O functions*.

C. I/O router and classifier

NVMetro implements various data paths for processing I/O requests received by the host. We adapt MDev-NVMe's queue shadowing [12], where NVMetro's I/O router receives commands from the guest using *virtual submission queues* (VSQ), and sends results to the guest using *virtual completion queues* (VCQ) (see Figure 1).

To inject custom logic into the kernel without compromising security, we selected eBPF as the platform for our I/O classifiers. These classifiers modify the request in two steps: firstly, *direct mediation*, where the classifier directly modifies a request's content. With direct mediation, each classifier can limit a command's privileges, e.g. by translating its requested logical block address (LBA) to the underlying device's real LBA (compared to MDev-NVMe which implements LBA translation directly inside its kernel module).

The second step is *request routing*, where requests are either routed by NVMetro or stopped by sending an error status to the VM's VCQ. NVMetro implements *iterative routing*, where a request traverses multiple hops following the classifier's policy. We manage iterative routing with a routing table that tracks each request's state during classification. To elaborate, requests are forwarded to the queue types corresponding to I/O paths shown in Figure 1: the fast path, which redirects requests to the underlying device's I/O queues, called the *host submission queues* (HSQ) and *host completion queues* (HCQ); the kernel I/O path, which sends requests through Linux's block device subsystem; or the notify path, which links to an UIF through *notify submission/completion queues* (NSQ/NCQ). Along with the VSQ, NVMetro's router worker threads actively poll the CQs of each path. We share these threads between multiple VMs in a round-robin fashion, and individually track each VM to stop polling them during inactivity.

When sending requests between components, NVMetro minimizes unnecessary memory copies even under long request paths. It only passes around each request's 64-byte command block, while the scatter-gather lists and data pages stay inside the VM's memory.

Finally, the I/O classifier can send one request to multiple targets simultaneously if needed. This is useful when the device and UIF need to work in parallel, e.g. during backup or mirroring. Moreover, the classifier can install additional *hooks* into the request. Hooks define certain events that happen during the request's lifecycle, e.g. when a request has been processed by hardware. Each hook calls the I/O classifier again to decide the next course of action until the request is satisfied.

D. Userspace I/O functions

UIFs are programs that processes each command from the notify path according to the storage function's requirements. Each UIF opens NSQs/NCQs as file descriptors, maps them into its address space using `mmap()` calls, and polls NSQs for requests from the I/O router. It also has access to the VM's memory to read and write request data. When the UIF finishes processing, it returns a status code to the kernel via the NCQ.

In our UIFs, we use an *adaptive polling* approach, where they can switch between active polling and OS-assisted waiting (using `epoll()`) depending on the activity level. This approach also permits serving multiple VMs using multiple UIFs in the same process to lower the CPU cost of busy polling.

To reiterate Section III-A, UIFs are meant to handle requests that should be further isolated, or cannot be easily implemented inside eBPF classifiers. For instance, as stated in Section II-B, eBPF programs run under multiple restrictions to ensure the kernel's integrity. Additionally, the time-critical, in-kernel nature of eBPF classifiers makes some features difficult to utilize (e.g. timers, memory allocations). In contrast, UIFs are free to choose the best APIs for fulfilling requests they receive. In other words, they can use basic `read()` and `write()` calls to serve data from a backend file, use `io_uring` to improve performance, or even send HTTP requests to a cloud service. However, to ease the creation of UIFs, we created an **UIF framework** that provides the following services:

1) Setting up notify queues and `io_uring` mappings for communication with the NVMetro router;
2) Configuring polling threads for I/O queues;
3) Parsing of incoming NVMe commands, as well as reading and writing of data pages from the VM;
4) Exposure of requests from the VMs as UIF events.

Our framework spans only 1100 lines of C++, and helps creating UIFs with minimal programming effort. We provide an example of an UIF under our framework in the next section.

IV. Use cases

In the following sections, we detail two examples of storage virtualization functions implemented with NVMetro: a function for transparent disk encryption, optionally integrating an Intel

SGX-based secure enclave; and a function that replicates data between two disks. For each storage function, we present its general request lifecycle and the roles of its components, namely the I/O classifier and accompanying UIF.

As stated above, our solution is exposed as a virtual NVMe controller inside each VM, with an additional control interface on the host. NVMetro storage functions are therefore managed by the system administrators by attaching each virtual controller to a namespace or partition on a backend NVMe device. We then use the control interfaces to insert eBPF classifiers and attach UIFs for storage functions they wish to use.

A. Transparent data encryption

We created a storage function to encrypt data on virtual disks, a critical feature for protecting sensitive data in cloud environments. Figure 2 shows the lifecycle of an I/O request under this function. In short, our eBPF classifier instructs the I/O router to send incoming I/O requests to an UIF, which decrypts and encrypts data during reads and writes respectively.

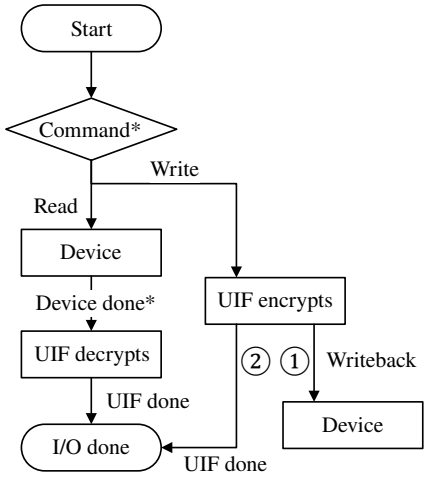

Fig. 2: Lifecycle of an I/O request with data encryption. The asterisk (*) denotes classifier invocation points.

I/O classifier. We specified two rules in our I/O classifier: (1) during reads: send the command to the physical disk, then once the disk read completes, forward it to the UIF for decryption; (2) during writes: send the command to the UIF for encryption, then forward it to the disk for writing. Our classifier runs at two critical decision points: once at the beginning of the request pipeline, and once more during a read command after the device finishes its read.

Listing 1 details the implementation of our encryption I/O classifier. The function `encryptor_classify` is our classifier's entry point, and is called every time the classifier is needed (see Figure 2). Each classifier is given an I/O context `ctx` that contains information about the current request. Depending on the request's processing stage (lines 13-24), the classifier must decide the next course of action:

- On a new request (`HOOK_VSQ`), the classifier reads the command's `opcode` (line 13). For reads, the classifier instructs the router to send this request to the device (`SEND_HQ`), and to invoke the classifier again when the device responds (line 16). For writes, the request is sent through the notify path (`SEND_NQ`). Once the UIF responds, the router immediately finishes the request without calling the classifier again (`WILL_COMPLETE_NQ`).
- When the aforementioned read finishes (`HOOK_HCQ`), the classifier checks the device's read error code. If an error occurred, this error is forwarded to the VM (line 8); otherwise, the read continues in the UIF (line 9).

Our example demonstrates both types of request modification available to a classifier:

- Direct mediation: by returning a NVMe status code (e.g. line 8, which forwards the physical device's status code). This status code is sent to the VM to stop the request;
- Request routing: The classifier chooses the target I/O paths to route our request. It can install a new hook (line 16) or automatically complete the request when its targets finish processing (lines 20 and 23).

Listing 1: Encryption eBPF classifier code.

```
1  int encryptor_classify(struct ctx *ctx) {
2    switch (ctx->current_hook) {
3    case HOOK_VSQ:
4      /* new request */
5      return encryptor_begin(ctx);
6    case HOOK_HCQ:
7      /* read device done, check for error */
8      if (ctx->error) return ctx->error | COMPLETE;
9      else return SEND_NQ | WILL_COMPLETE_NQ;
10   }
11 }
12 int encryptor_begin(struct ctx *ctx) {
13   switch (ctx->cmd.common.opcode) {
14   case nvme_cmd_read:
15     /* read commands that need reading ciphertext */
16     return SEND_HQ | HOOK_HCQ | WAIT_FOR_HOOK;
17   case nvme_cmd_write:
18     /* write commands that need encrypting,
19      * UIF will finish the command */
20     return SEND_NQ | WILL_COMPLETE_NQ;
21   default:
22     /* send to device */
23     return SEND_HQ | WILL_COMPLETE_HQ;
24   }
25 }
```

Userspace I/O function. Our encryption UIF performs three tasks: (1) in-place decrypting of ciphertext from the physical device; (2) encrypting of plaintext from the guest into a temporary buffer; and (3) writing of ciphertext from step (2) to disk with `io_uring`. Our encryptors use the standard XTS-AES algorithm and are compatible with Linux's `dm-crypt`.

Listing 2: Request processing code of encryption UIF.

```
1 bool uif::work(nvme_cmd cmd, u32 tag, u16 &status) {
2   switch (cmd.common.opcode) {
3   case nvme_cmd_read:
4     status = do_read(cmd);
5     return false; /* respond with status */
6   case nvme_cmd_write:
7     do_write_async(cmd, tag);
8     return true; /* asynchronous response later */
9   }
```

```
10 }
11 u16 uif::do_read(nvme_cmd cmd) {
12   for (auto data=parse(cmd); !data.at_end(); data++)
13     if (!decrypt(*data, data.lba())) /*inplace*/
14       throw std::runtime_error("cannot decrypt");
15   return NVME_SC_SUCCESS;
16 }
17 void uif::do_write_async(nvme_cmd cmd, u32 tag) {
18   auto data = parse(cmd);
19   auto ticket = new iovec_ticket({.tag = tag});
20   auto buf = malloc(data.nbytes());
21   /* encrypt data into temporary buffer */
22   for (; !data.at_end(); data++) {
23     auto block = buf.subspan(data.block_offset(),
         data.lba_size());
24     if (!encrypt(/*out*/ block, /*in*/ *data, data.
         lba()))
25       throw std::runtime_error("cannot encrypt");
26   }
27   /* write to disk from the UIF with io_uring */
28   ticket->iovecs.push_back({buf, data.nbytes()});
29   queue_writev(ticket, data.disk_addr());
30 }
```

Listing 2 shows an abbreviated version of our UIF code. Each UIF is represented by a C++ class (`uif`) following our implementation interface. Our framework passes incoming requests to the UIF's `work` function, which classifies the request's type (lines 2-9). During reads, the implementation is straightforward: the UIF iterates over the data blocks coming from the device (line 12), then decrypts them in-place (line 13) and signals the VM of a successful decryption (line 15). During writes, the UIF allocates a temporary buffer (line 20) which is used for encryption (lines 22-26). The temporary buffer is written to disk with `io_uring` (lines 28-29) and the request completes when this write finishes. As seen from the code snippet, our UIF framework takes care of queue handling, request and memory management, while the UIF code only needs to encrypt and decrypt data. Moreover, our UIF framework supports all C++ features and libraries, making UIF development simpler than that of Linux kernel modules.

We implemented two encryption UIFs using the same classifier: one normal UIF, and one using Intel SGX enclaves. Both versions use AES-NI instructions for encryption, the same as `dm-crypt`, SPDK and other encryption software. Our SGX-based UIF stores the cryptographic key inside a hardware enclave. Both UIFs share substantial amounts of code, except for only ≈ 120 lines of SGX-specific code.

B. Live disk replication

We created a mirroring UIF that replicates data between two NVMe drives: a primary drive attached directly to the local host, and a secondary drive attached to a remote host. The two hosts are connected together using NVMe over Infiniband.

Our I/O request pipeline is as follows: our classifier passes read requests directly from the guest to the primary disk, while write requests are sent to both the primary disk and UIF. The UIF then forwards the write request to the secondary disk using `io_uring`. The mirroring process is synchronous, where writes are not completed until both the local and remote disks finish the request, thus allowing easy reuse of the VM's data buffers.

C. Implementation effort

As stated in Section III-D, our UIF framework aids the implementation of fast and simple storage UIFs. Table I shows a breakdown of the lines of code needed for each of our storage function. Note that our normal and SGX encryptor functions share the same classifier and 80% of UIF code.

TABLE I: Source code sizes of NVMetro classifier and UIF implementations.

Function	Component	Lines
Encryptor	Classifier	32
Encryptor	Normal UIF	520
Encryptor	SGX UIF + enclave	501
Replicator	Classifier	16
Replicator	UIF	307
Framework	—	1116

V. Evaluation

Our goal for the evaluation of NVMetro is twofold:

1) Compare the I/O performance of NVMetro to existing solutions in the basic use case;
2) Show our UIF framework's flexibility and ease of use through various real-world storage function use cases.

A. Experimental setup

We evaluate the performance of NVMetro using our both UIFs presented in Section IV with multiple different workloads: firstly, benchmarks of I/O performance under various configurations with `fio` [13]; and secondly, database evaluations using the YCSB suite [14].

We use two platforms for our evaluations: two Dell PowerEdge R420 servers, each equipped with 2x Intel Xeon E5-2420 v2 and 48 GB of RAM for most evaluations; and a Dell Precision 7540 laptop with an Intel Core i5-9400H and 16 GB of RAM for disk encryption evaluations with Intel SGX. Each machine is equipped with a Samsung 970 EVO Plus 1TB SSD for evaluation purposes. Experiments are conducted inside a QEMU VM with 6 GB of RAM and 4 physical cores (servers)/2 physical cores (laptops) running Ubuntu 20.04.

`fio` evaluation setup. To evaluate NVMetro's raw performance, we executed `fio` while varying the I/O block sizes, benchmark modes (random, sequential, read/write/mixed), queue depths (QD), and number of parallel jobs. We ran each experiment 3 times, and recorded the resulting average I/O per second (IOPS). We measured each experiment's whole-system CPU consumption to compare the solutions' performance impacts. Table II shows a detailed list of configurations.

We also evaluated the latency of various storage solutions. We test each solution at a fixed rate of $10,000$ IOPS, while varying the block sizes and queue depths, and report the median and 99^{th}-percentile latencies for each configuration.

YCSB evaluation setup. We benchmarked NVMetro using the YCSB suite's 6 built-in workloads (version `ce3eb9c`). We configured each workload to run on RocksDB over ext4; to minimize filesystem overhead, we disable the journal, discards

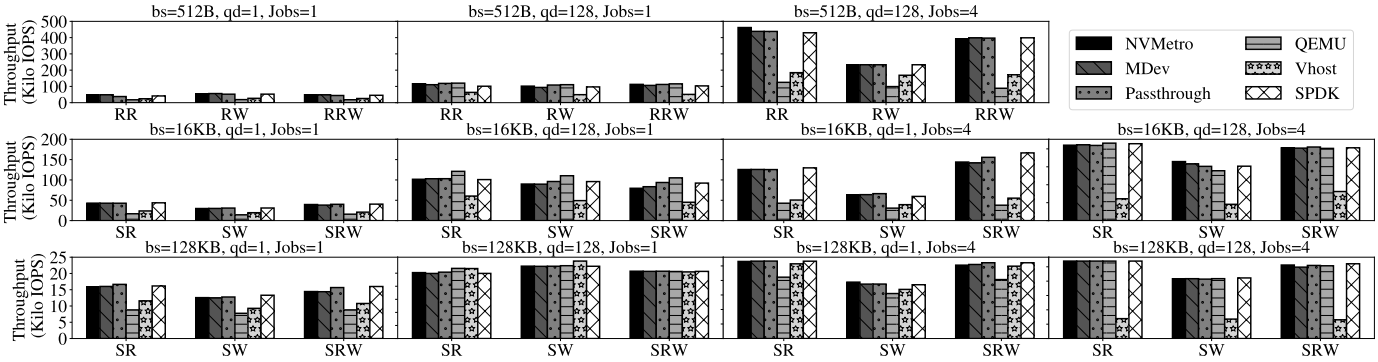

Fig. 3: Basic evaluations: `fio` performance for each workload configuration and storage virtualization method.

TABLE II: List of `fio` benchmark configurations.

Block size	Mode	QD	Nr. jobs
512	Random read (RR)	1, 128	1
512	Random write (RW)	1, 128	1
512	Mixed random R/W (RRW)	1, 128	1
512	Random read (RR)	128	4
512	Random write (RW)	128	4
512	Mixed random R/W (RRW)	128	4
16K	Sequential read (SR)	1, 128	1, 4
16K	Sequential write (SW)	1, 128	1, 4
16K	Mixed sequential R/W (SRW)	1, 128	1, 4
128K	Sequential read (SR)	1, 128	1, 4
128K	Sequential write (SW)	1, 128	1, 4
128K	Mixed sequential R/W (SRW)	1, 128	1, 4

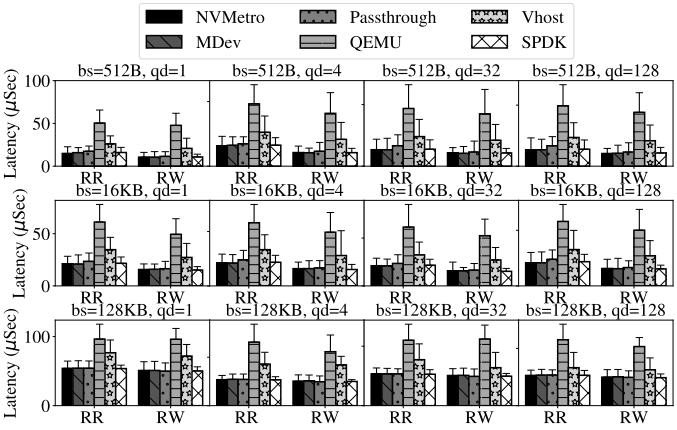

Fig. 4: NVMetro latency evaluation results. Columns denote median latency; 99th-percentile latency is shown in whiskers.

and access time features. We ran each workload 3 times with 1 million operations each on a dataset of 3 million records. We evaluate two scenarios: 1) one YCSB job on 1 DB instance; and 2) four parallel jobs, each with its own DB instance.

B. Basic performance evaluations

In this section, we compare the overhead of NVMetro with other storage solutions: direct PCIe passthrough; MDev-NVMe (implemented by Maxim Levitsky [15]); paravirtualized disk with in-kernel `vhost-scsi`; virtual disk using QEMU's `virtio-blk` with `io_uring`; and finally, SPDK's `vhost-user`-based `virtio-blk`. NVMetro uses a dummy eBPF classifier without UIF.

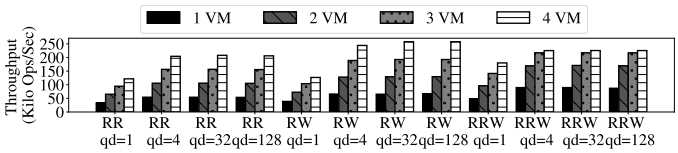

Fig. 5: NVMetro scalability evaluation results.

Figure 3 shows the performance of NVMetro compared to other solutions in the `fio` benchmark. In all configurations, NVMetro with a dummy eBPF classifier performs similarly to MDev-NVMe, SPDK and device passthrough. Being userspace-based, QEMU's `virtio-blk` performs significantly worse than NVMetro at higher I/O rates and lower queue depths; for example, NVMetro is 2.7× faster at 512B RR than QEMU at QD1/1 job. QEMU regains performance at higher QDs, potentially due to it redistributing I/O requests across multiple worker threads; in fact, QEMU at 16K/QD128/1 job performs the best, being between 19% to 32% faster than NVMetro. In comparison, `vhost-scsi` despite being in-kernel falls behind in performance, being one of the worst performers regardless of benchmark configuration.

Figure 4 shows the request latency figures with `fio`, where the bar heights represent the median latencies while the whisker heights represent 99$^{\text{th}}$-percentile latencies. Among our tested configurations, a pattern emerges where NVMetro, MDev-NVMe and SPDK, being polling-based, share approximately the same median and tail latencies. Direct PCIe passthrough without polling falls behind with a median latency 18.2% higher than NVMetro at 512B RR and 9.1% higher at 512B RW, potentially due to the overhead of forwarding device interrupts to the guest. Vhost exhibits poor latencies even at our low I/O rate, namely 73.6% higher at 512B RR and 97.6% higher at 512B RW. QEMU's virtual storage again performs even worse, with 3.4× higher median random read latency and 4.1× higher write latency at 512B. Concerning tail latencies, the only solution with a lower 99$^{\text{th}}$-percentile write latency than NVMetro is SPDK, at 5.9%, 18.0% and 13.0% for 512B, 16K and 128K blocks.

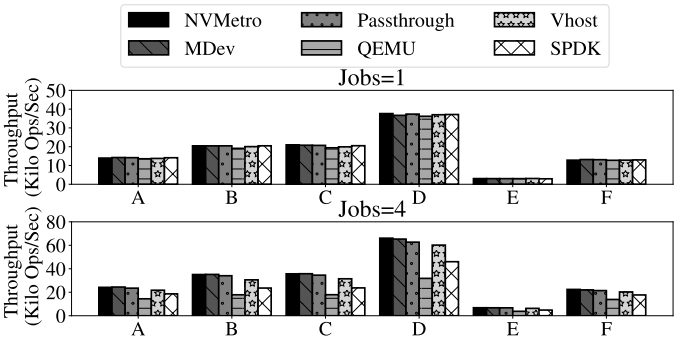

Fig. 6: YCSB throughput for each workload type (A-F).

Figure 5 shows NVMetro's scalability under an increasing number of small VMs. Each VM is given 2 GB of RAM, 1 dedicated physical core, and a dedicated partition on a shared NVMe namespace.[1] We set up NVMetro to use one host kernel thread to concurrently serve all VMs. All evaluations were performed at a block size of 512B. We observe that system throughput gradually increases as we add more VMs, confirming NVMetro's scalability even with high VM densities.

Our YCSB benchmark results in Figure 6 show little performance variation between all solutions with 1 running job. At 4 parallel jobs, YCSB becomes more I/O-bound and therefore shows more performance variations, while MDev-NVMe and NVMetro stay close to native passthrough performance (within approximately 3%). Other solutions fall behind, with `vhost-scsi`, SPDK and QEMU being up to 10%, 31% and 49% slower than device passthrough respectively.

C. Disk encryption evaluations

In this section, we demonstrate the performance of disk encryption using NVMetro (with and without SGX) compared to Linux's `dm-crypt` and `vhost-scsi` as the virtual storage interface. We also make comparisons with the unencrypted scenarios presented above. Our non-SGX UIF uses 2 threads; our SGX UIF uses 1 worker + 1 SGX switchless thread.

Overall, Figure 7 shows that our non-SGX UIF outperforms `dm-crypt` at all presented configurations. Notably, at (512B, 16K, 128K)/QD1/1 job, our UIF is up to 1.6×, 1.5× and 1.4× faster than `dm-crypt`. Our solution is even faster with higher parallelism, being 3.2× faster with 16K reads/QD128/4 jobs and 3.7× faster at 128K. Meanwhile, our SGX-based encryption UIF performs mostly the same as non-SGX, excepting 16K/QD128/4 jobs and 128K/QD128/4 jobs being up to 50% and 75% slower than non-SGX on average, and 128K SW/QD128/4 jobs being 45% slower than `dm-crypt`. These results are explained by its lower encryption thread count (as it uses 1 thread for switchless calls).

In Figure 8, the YCSB benchmark shows similar performance between our non-SGX UIF and `dm-crypt`. However, when varying the YCSB job count, we observe a slight performance gap between SGX and non-SGX. With one YCSB job, our SGX UIF is up to 35% slower than non-SGX in workload D; however, it gains back some ground at 4 jobs, with the worst-performing workload D only losing 21% performance, while most other workloads become comparable to non-SGX.

[1]Note that our smaller VM size in this experiment prevents direct comparison with the throughput evaluations presented above.

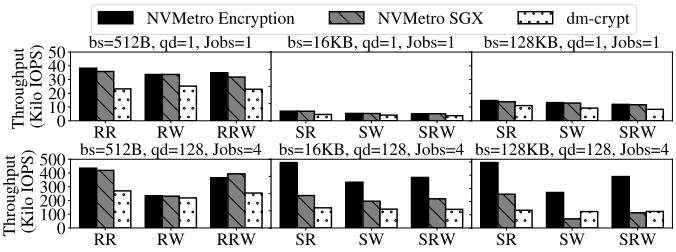

Fig. 7: Disk encryption evaluations with `fio`.

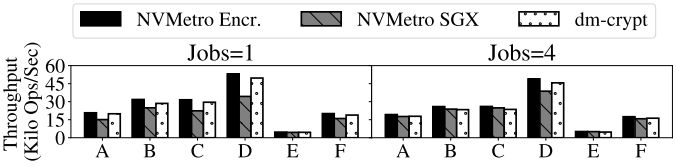

Fig. 8: Disk encryption evaluations with YCSB.

D. Disk replication evaluations

In this section, we compare NVMetro's disk mirroring with Linux's `dm-mirror`+`vhost-scsi` on the VM host. In general, both NVMetro and `dm-mirror` perform better at reading than writing; this is easily explained since reads can be directly serviced by the local drive without propagating to the remote. When comparing the two solutions using `fio` (see Figure 9), NVMetro outperforms `dm-mirror` at all configurations by 68%, 220% and 291% at 512B reads/QD1/1 job, 512B reads/QD128/4 jobs and 128K reads/QD128/4 jobs respectively, demonstrating NVMetro's I/O path flexibility in choosing the more efficient data read path. Figure 10 shows our disk replication performance in the YCSB benchmark. In general, NVMetro is faster than `dm-mirror` no matter the workload or number of parallel jobs. We again see our scalability advantages: NVMetro performs 2% better at workload D with 1 YCSB job but 17% better with 4 jobs.

E. Overhead evaluations

In this section, we compare the CPU usage of each virtualization method while running `fio` under each scenario presented above. The CPU usage is presented in terms of total system CPU time, including the VM and any host agents.

Basic evaluations (Figure 11). Device passthrough predictably performs the best among all tested configurations. MDev-NVMe, NVMetro and QEMU perform similarly, using $\approx 85\%$ more total CPU than passthrough at 512B/QD1/1 job, and $\approx 26\%$ more in the intensive benchmark of 512B/QD128/4 jobs; with the exception of 128KB/QD1/1 job where QEMU uses less CPU than the other two. `vhost-scsi` is more efficient still, being the second-lowest CPU-consuming virtualization method, only bested by device passthrough. Conversely,

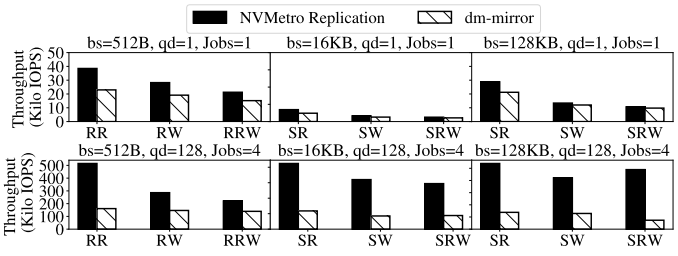

Fig. 9: Disk replication evaluations with `fio`.

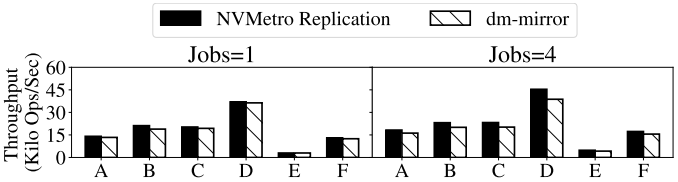

Fig. 10: Disk replication evaluations with YCSB.

SPDK uses the most CPU time, with a $\approx 56\%$ overhead at 512B/QD128/4 jobs. The higher CPU usage of MDev-NVMe, NVMetro and especially SPDK is explained by these solutions using active polling to process I/O requests.

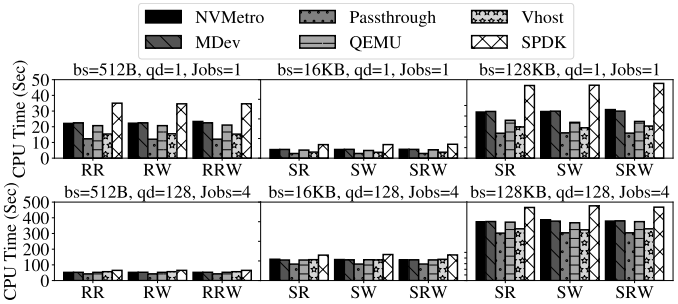

Fig. 11: CPU consumption of `fio` with basic evaluation.

Disk encryption (Figure 12). At (512B, 16K, 128K) QD1/1 job, our encryption UIF uses around $2.7\times$, $2.4\times$ and $2.1\times$ the CPU of `dm-crypt`. While our UIF's CPU utilization is higher than that of `dm-crypt` at lower parallelism, we gain ground in performance and CPU usage at higher parallelism: at 4 parallel jobs, NVMetro uses around the same CPU time as `dm-crypt` in reads, and even slightly less at 16K and 128K.

Our SGX-based UIF has a rather uniform CPU cost at lower parallelisms: with (512B, 16K, 128K)/QD1/1 job, we use $\approx 10\%$ and 12% more CPU for essentially the same performance. At QD128/4 job, our UIF uses the same amount of CPU due to our maximum CPU constraint.

Disk replication (Figure 13). At 512B/QD1/1 job, 512B/QD128/4 jobs and 128K/QD128/4 jobs, NVMetro incurs a CPU cost up to 178%, 36% and 76% higher than `dm-mirror`; nevertheless, this CPU cost is coupled with better performance, especially at 128K reads/QD128/4 jobs where we pay 35% more CPU for 291% more throughput, a combination of NVMetro's poll-based I/O and efficient request routing.

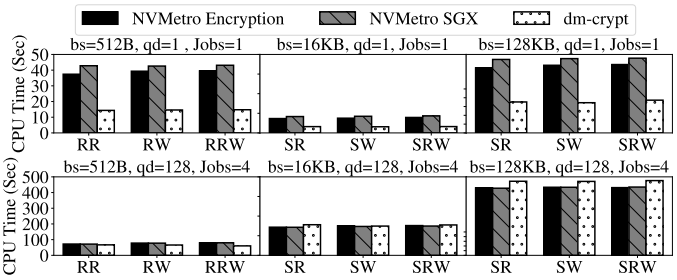

Fig. 12: CPU consumption of `fio` with disk encryption.

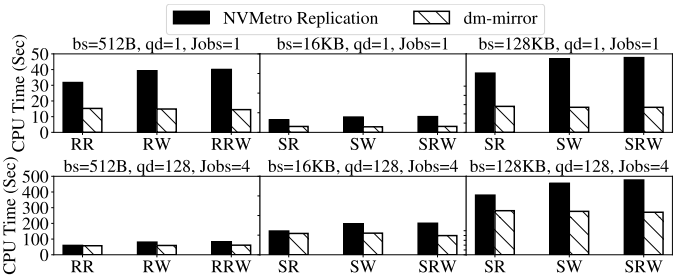

Fig. 13: CPU consumption of `fio` with disk replication.

F. NVMetro's flexibility and ease of use in perspective

As we claimed in Section III-B, NVMetro's storage framework is more flexible and easier to use than existing systems. In this section, we support our claims by analyzing our storage function implementations in contrast to other storage solutions.

Compared to Linux's `vhost-scsi` and device mapper. Linux's in-kernel storage virtualization involves two components: the `vhost-scsi` facility that provides a virtual SCSI interface to VMs, and a device mapper ("DM" for short) that provides a stackable logic layer on top of storage devices (similar to FreeBSD's GEOM [16]). Together, these two give the host control over each VM's storage access.

Linux's device mapper implements its mapping targets inside the kernel, rather than as independent programs. These targets can be stacked in order to combine simple block mapping functions; however, the use of specific technologies such as Intel SGX poses an additional challenge, as Linux only supports user-mode SGX applications at the moment. In contrast, we easily integrated Intel SGX into our encryption UIF.

NVMetro is designed from the ground up for fast kernel-UIF communication using multiple asynchronous queues and adaptive polling. Furthermore, NVMetro's userspace-kernel decoupling lets storage functions serve multiple VMs while reducing the use of costly I/O polling threads. Finally, our request router's eBPF-coded fast paths help reduce the cost of mediation, as apparent from our disk replication implementation: the UIF only needed to consider writes, while reads are filtered out by our classifier and directly passed to disk.

Compared to MDev-NVMe. To reiterate, MDev-NVMe serves as a basis for our implementation of NVMetro. As such, our goal is not to beat MDev-NVMe in raw performance; instead, NVMetro brings an innovative classification and routing

component, and a fast pathway for UIFs to communicate with its VMs. Our evaluations showed that these components did not introduce a significant overhead compared to the existing MDev-NVMe mechanism. A possible alternative is to implement all of the storage logic directly inside the MDev-NVMe module, or to offload it to the DM layer; however, these approaches have the same limitations as other in-kernel solutions.

Compared to in-VMM virtualization. Userspace VMMs such as QEMU have direct access to a VM's execution states and virtual devices. As such, they have full control over a VM's I/O request flow. However, they also have two significant limitations. Firstly, virtual I/O needs to be trapped in the kernel then relayed back to the VMM. Afterwards, more hypervisor operations are needed to signal the VM of I/O status (e.g. using virtual interrupts), and to resume VM execution after a trap. Secondly, even with solutions that avoid the above flaw (e.g. Virtio at high QDs), each VMM needs to handle its own VM's storage requests. With high VM densities, handling I/O separately on each VM wastes large amounts of CPU time and context switches, thus limiting the scalability of this solution.

Compared to SPDK. SPDK is comparable to NVMetro as a set of tools for writing user-mode storage applications. Both possess similar capabilities: stackable storage logic, colocating multiple storage targets in one process, and so on. However, NVMetro provides two main benefits compared to SPDK. Firstly, NVMetro does not require exclusive assignment of a storage device; the host and multiple VMs can easily share one device at the same time (e.g. accessing different partitions on the same disk; or in a shared-disk filesystem scenario). Secondly, NVMetro can be gradually applied to I/O requests as requirements evolve. Particularly, the storage developer does not need to consider hardware internals, or the handling of irrelevant requests and commands; relevant requests are selected in eBPF, and our UIFs communicate with our router using standard POSIX APIs.

VI. Related works

General computational storage architecture. SNIA's Computational Storage Architecture and Programming Model [17] defines a general structure of computational storage applications, where different kinds of storage engines (e.g. eBPF-based) can be embedded into various device classes. It also defines several types of computational storage functions for these engines.

Virtual storage providers. SPDK [3] is a fast storage framework based on top of the NVMe protocol. In the same vein as DPDK, it uses an userspace driver via device passthrough to deliver various virtual I/O services. Vhost is Linux's paravirtualized device framework based on the Virtio specification for fast and efficient I/O services for KVM guests. It offloads I/O processing to the host kernel [18] or an external process (e.g. SPDK) via `vhost-user` [19], [20]. MDev-NVMe [12] describes a NVMe virtualization layer based on active polling to improve I/O throughput and reduce latency. Notably, MDev-NVMe bypasses many subsystems of the Linux kernel to reduce the cost of each I/O operation. FAST I/O [21] proposes QoS service controls of I/O on NVMe devices by submitting high-priority requests directly to an admin NVMe queue, therefore bypassing the operating system-level queues, and by writing request data to the Host Memory Buffer (HMB) region.

NVMetro also belongs to the category of virtual storage providers. The advantage of NVMetro compared to others is a combination of kernel- and userspace-based logic to allow developers to quickly and easily customize their virtual I/O path per-request depending on their use case.

Sandboxed-bytecode (eBPF, WebAssembly)-based solutions. Most works in this category propose offloading computing tasks to local storage agents. Zhong et al. [8] investigate the feasibility of inserting BPF hooks into Linux's storage stack to provide extra functionalities, e.g. tree lookups. Griffin [22] envisions an API set using eBPF to add logic to storage apps running on edge computing nodes. Kourtis et al. [23] follow in the same line by running eBPF on top of NBD, and propose ways to use eBPF for KV store and SQL offloading.

Generally speaking, these solutions suggest extending eBPF or replacing it with another runtime (e.g. WebAssembly), citing eBPF's current limitations. In contrast, NVMetro requires no change to the kernel's eBPF implementation, as the eBPF code only serves as a first-line classifier inside the request router; complex operations can be offloaded to UIFs.

Hardware-based solutions. In this category, LeapIO [24] presents a new storage stack that offloads virtualization tasks onto on-disk processors coupled with smart memory and NIC sharing to improve performance. FastPath [25] adds a FPGA-based computing engine between the host and storage device, then exposes an API to offer a fast path to applications needing high I/O performance. FastPath_MP [26] extends FastPath with support for multiple I/O queues to take advantage of the parallelism offered by NVMe devices. Similarly, FVM [27] interposes NVMe devices with FPGA to virtualize storage using I/O queue emulation and storage address translation.

VII. Conclusion

In this paper, we introduced NVMetro, a flexible I/O virtualization framework that eases the development of sophisticated storage functions. NVMetro builds upon a mediated NVMe interface with a combination of fast eBPF-based I/O classifier/router and userspace I/O functions. By allowing the creation of multiple I/O paths of varying characteristics, NVMetro ensures that storage function remains fast, secure and manageable regardless of the use case. We described the design criteria that lead to NVMetro's features, and elaborated on the development of several sample storage use cases. We evaluated NVMetro in comparison to existing systems, and showed the performance, scalability and simplicity of our storage function implementations using multiple benchmarks, thus demonstrating the flexibility of our framework.

Acknowledgments

This work is supported by the French *Agence nationale de la recherche* under the ANR WalkIn (20-CE25-0005) and LabEx

CIMI (11-LABX-0040) projects.

REFERENCES

[1] NVM Express, Inc., "NVM Express specifications," https://nvmexpress.org/specifications/, 2021.

[2] Intel Corporation, "Intel Optane SSD P5800X Series," https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html, 2021.

[3] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, "SPDK: A development kit to build high performance storage applications," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 154–161.

[4] DPDK Project, "Kernel NIC interface," https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html, 2021.

[5] M. Szeredi, "FUSE: Filesystem in userspace," https://github.com/libfuse/libfuse, 2010.

[6] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or not to FUSE: Performance of user-space file systems," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 59–72.

[7] Cloudflare, "Why we use the Linux kernel's TCP stack," https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/, 2016.

[8] Y. Zhong, H. Wang, Y. J. Wu, A. Cidon, R. Stutsman, A. Tai, and J. Yang, "BPF for storage: an exokernel-inspired approach," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, 2021, pp. 128–135.

[9] S. McCanne and V. Jacobson, "The BSD packet filter: A new architecture for user-level packet capture." in *USENIX winter*, vol. 46, 1993.

[10] The kernel development community, "BPF documentation - the Linux kernel documentation," https://www.kernel.org/doc/html/latest/bpf/index.html, 2021.

[11] Cilium, "BPF and XDP reference guide," https://docs.cilium.io/en/latest/bpf/, 2021.

[12] B. Peng, H. Zhang, J. Yao, Y. Dong, Y. Xu, and H. Guan, "MDev-NVMe: A NVMe storage virtualization solution with mediated pass-through," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 665–676.

[13] J. Axboe, "Flexible I/O tester," https://github.com/axboe/fio, 2021.

[14] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[15] M. Levitsky, "NVME VFIO mediated device," https://lkml.org/lkml/2019/3/19/458, 2019.

[16] P.-H. Kamp, "GEOM tutorial," https://papers.freebsd.org/2004/phk-geom-tutorial.files/bsdcan-04.slides.geomtut.pdf, 2004.

[17] SNIA, "Computational storage architecture and programming model," https://www.snia.org/sites/default/files/technical_work/PublicReview/SNIA-Computational-Storage-Architecture-and-Programming-Model-0.8R0-2021.06.09.pdf, 2021.

[18] Red Hat, Inc., "Deep dive into Virtio-networking and vhost-net," https://www.redhat.com/en/blog/deep-dive-virtio-networking-and-vhost-net, 2019.

[19] ——, "A journey to the vhost-users realm," https://www.redhat.com/en/blog/journey-vhost-users-realm, 2019.

[20] Z. Yang, C. Liu, Y. Zhou, X. Liu, and G. Cao, "SPDK Vhost-NVMe: Accelerating I/Os in virtual machines on NVMe SSDs via user space Vhost target," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*. IEEE, 2018, pp. 67–76.

[21] K. Kim, S. Kim, and T. Kim, "FAST I/O: QoS supports for urgent I/Os in NVMe SSDs," in *Proceedings of the 2020 5th International Conference on Intelligent Information Technology*, 2020, pp. 146–151.

[22] G. Frascaria, A. Trivedi, and L. Wang, "A case for a programmable edge storage middleware," *arXiv preprint arXiv:2111.14720*, 2021.

[23] K. Kourtis, A. Trivedi, and N. Ioannou, "Safe and efficient remote application code execution on disaggregated NVM storage with eBPF," *arXiv preprint arXiv:2002.11528*, 2020.

[24] H. Li, M. Hao, S. Novakovic, V. Gogte, S. Govindan, D. R. Ports, I. Zhang, R. Bianchini, H. S. Gunawi, and A. Badam, "LeapIO: Efficient and portable virtual NVMe storage on ARM SoCs," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 591–605.

[25] A. Stratikopoulos, C. Kotselidis, J. Goodacre, and M. Luján, "FastPath: towards wire-speed NVMe SSDs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 170–1707.

[26] ——, "FastPath_MP: Low overhead & energy-efficient FPGA-based storage multi-paths," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 17, no. 4, pp. 1–23, 2020.

[27] D. Kwon, J. Boo, D. Kim, and J. Kim, "FVM: FPGA-assisted virtual device emulation for fast, scalable, and flexible storage virtualization," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 955–971.