Virtual NVMe-based Storage Function Framework with Fast I/O Request State Management

Tu Dinh Ngoc, Boris Teabe, Georges Da Costa, Daniel Hagimont

***Abstract*—Current cloud environments provide numerous storage functions to virtual machines such as disk encryption, snapshotting, compression and so on. These functions are implemented using software stacks inside the hypervisor's kernel, emulator, or as a userspace polling driver like SPDK. However, each stack brings its own limitations: Linux's kernel I/O stack cannot easily integrate proprietary technologies such as Intel SGX, while SPDK requires significant changes in software development and tooling yet lacks the rich feature set of existing solutions like Linux LVM. To remedy these limitations, we introduce NVMetro, a high-performance storage framework for virtual machines based on the NVMe protocol. NVMetro provides multiple I/O paths that can be dynamically combined to fit the needs of each storage function. It links these paths together with an eBPF-based I/O router/classifier framework, as well as a userspace software stack for out-of-kernel I/O processing. We implemented three different storage functions with NVMetro and evaluated them under various workloads. Our results show that NVMetro approaches the performance of kernel-bypass solutions like SPDK while maintaining the compatibility and ease of use of in-kernel storage stacks.**

***Index Terms*—Storage virtualization, kernel bypass, multipath, eBPF, NVMe, sandboxed I/O.**

I. Introduction

Virtualization is extensively used in the modern datacenter as a security boundary to isolate workloads such as virtual machines (VMs). On top of virtualization, cloud providers offer numerous storage services to the customer via *storage functions*. For example, VM disk snapshotting is a standard feature on most virtualization platforms. Cloud VMs also come encrypted by default for regulatory and data privacy reasons. These storage functions are often implemented either in the VM emulator (e.g. QEMU), using standard Linux I/O mechanisms; directly in the hypervisor kernel (e.g. Linux Vhost); through a kernel-bypass driver (e.g. SPDK); or finally by directly assigning a storage device to the VM via SR-IOV (aka. device passthrough).

We observe several tradeoffs in various properties among these four choices. For example, the first two options are the most flexible for the cloud provider, as the tooling for managing and monitoring them are widely understood. However, their performance tends to be inferior to other solutions, especially considering recent rapid increases in storage hardware performance. Namely, Peng et al. [1] showed that Virtio storage as provided by QEMU only reaches 50% of native throughput, leaving plenty of performance on the table. In the same vein, Zhong et al. reported that 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 [2]. Some userspace solutions like FUSE also come with issues that can severely degrade performance [3]. These solutions also show differences in their security, with userspace and passthrough solutions being more isolated than in-kernel ones.

A common concern in choosing a cloud storage framework is its level of compatibility with existing tools and applications. 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). These drawbacks are reasons to choose in-kernel I/O implementations over a higher-performing userspace or hardware-based one [6], as they can readily take advantages of Linux's drivers and I/O features.

However, in-kernel stacks are not a complete guarantee of compatibility or stability. Firstly, integrating a new technology or programming language inside the kernel requires a new kernel-specific SDK, which is a considerably more challenging task than building a userspace application. 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, Linux does not yet support creating an SGX enclave directly inside the kernel. The same applies to writing storage functions in a language not supported in the Linux kernel such as C++ or Go. Moreover, in the case of Linux, its in-kernel API is unstable, meaning in-kernel stacks are faced with constant changes, leading to costly refactoring and constant breakages. Finally, implementing sophisticated logic directly inside the kernel increases its attack surface and reduces its reliability, since any bug inside the storage function can lead to host crash or compromise, as shown by various vulnerabilities in Linux's virtualization components [7]–[9].

To reiterate, each current storage function platform presents its own tradeoffs in performance, flexibility or compatibility. Our insight is as following: rather than locking storage functions to a single communication API, no matter how advanced, we instead aim to provide them with multiple access methods that can be switched on-the-fly, trusting the storage developer to choose the correct method based on their application requirements. For this purpose, we introduce *NVMetro* (initially presented in [10]), our framework for creating storage functions with a focus on flexible choice in I/O paths. NVMetro presents a virtual NVMe controller architecture that sits between the host and the guest to control the flow of guest I/O requests. To accomplish our design criteria, NVMetro focuses around three main elements: (1) multiple *I/O paths* for handling virtual storage requests, varying in their properties wrt. the storage

function; (2) an *I/O router* supporting pluggable, iterative *request classifiers* based on Linux's *Extended Berkeley Packet Filter* (eBPF) for encoding custom logic into the storage virtualization pipeline; and (3) a kernel-user API that assists the creation of *userspace I/O functions* (UIFs) for high-performance storage processing. Together, these components facilitate the creation of complex storage functions in a modular fashion.

In particular, we detail the implementation of three use cases: a disk snapshotting function, a data encryption function, and a data replication function. Our disk snapshotting function features a decoupled architecture that combines a block address translation layer running in userspace with a fast in-kernel translation cache for accelerating I/O operations, meant to demonstrate a storage function with complex classifier-UIF cooperation. Our data encryption function resolves the aforementioned drawbacks of implementing storage logic in the kernel with two parallel UIF implementations, with and without Intel SGX integration.

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 design and implementation of NVMetro's I/O routing, classification and UIF components. Section IV investigates three NVMetro use cases in depth to demonstrate how NVMetro helps implementing simple yet efficient caching and mediation of storage requests. We evaluate these use cases in Section V to show their performance under various workloads. Our results demonstrate that NVMetro takes good advantage of the storage performance of modern hardware, with our NVMetro-based storage functions being equal to or better than competing solutions while using less CPU time. 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; the *zoned namespace command set* for devices that require a particular write pattern for optimal performance; 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 supports 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) [11] 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 [12].

The BPF instruction set was enhanced in Linux into *Extended BPF* (eBPF) with extra instructions and registers. Before running each eBPF program, the Linux kernel verifies its safety through various properties, including constraints on memory accesses, loops and program size. The Linux kernel contains a verifier to validate the safety of submitted eBPF programs, as well as a JIT compiler to optimize filter execution [13]. Depending on their type, eBPF programs can call various kernel helper functions, including functions that implement *eBPF maps*, which permit them to store data and communicate with other programs.

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 [14].

III. Design and implementation of NVMetro

In the following sections, we describe the design of NVMetro's core components and the relations between I/O paths.

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. In other words, they are "all-or-nothing": once a storage function's business logic has been

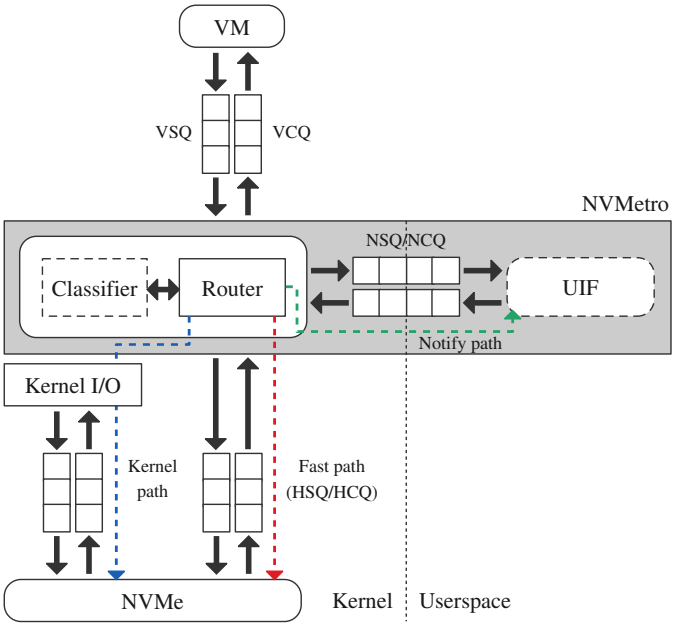

Fig. 1. NVMetro architecture and I/O paths. Customizable components are drawn in dashed outline. VSQ/VCQ, HSQ/HCQ and NSQ/NCQ denote *virtual*, *host* and *notify* submission and completion queues (see Section III-B).

implemented (inside or outside the kernel), the way it is implemented cannot be easily changed, e.g. by switching to a different API or adding an early request processing stage. In contrast, NVMetro gives 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, 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.

As stated in Section I, one of NVMetro's key contributions is its multipath I/O design, where each I/O request from the guest can travel to its target through one of possible preset paths. Yet unlike other solutions, NVMetro does not use I/O multipathing simply for redundancy or high-availability; it explicitly exposes these paths to the storage function on a per-request basis. Figure 1 showcases the different I/O paths in our architecture. In short, requests from the VM pass through a central *I/O router*, controlled by a customizable, sandboxed *I/O classifier* to route requests through one of three *I/O paths*: a *fast path* to a physical NVMe device (red arrow); a *kernel path* (in blue); and a *notify path* (in green) to an external userspace I/O function (UIF).

Each I/O path presents various tradeoffs that the classifier must take into account. The fast path, our simplest I/O path, sends 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. Finally, NVMetro's *notify path* exports requests for processing outside of the host kernel. Said requests are handled by UIFs making up part of the desired storage function. We suggest using UIFs when in-kernel request processing is complex to implement, or when extra isolation of the storage function is required.

A key point of NVMetro is that its 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.

The next sections detail NVMetro's *I/O router and classifier*, and its accompanying *userspace I/O functions*.

B. I/O router and classifier

To recall, Figure 1 showed the various data paths that NVMetro implements for processing I/O requests received by the host. We adopt MDev-NVMe's queue shadowing and adaptive polling [1], where NVMetro's I/O router receives commands from the guest using *virtual submission queues* (VSQ), and sends result to the guest using *virtual completion queues* (VCQ). The router continuously polls these queues for new requests to be inspected by the classifier, while going to sleep during periods of low activity to reduce CPU usage.

NVMetro defines a clear request mediation model for I/O classifiers. They have control over an I/O request by *direct mediation*, where the classifier directly modifies a request's content; and/or by *request routing*, where the request is redirected to another destination. We implement these controls by running the classifiers as eBPF programs inside the NVMetro kernel, then passing their output to the NVMetro router for further decision. In this fashion, the router keeps tight control over classifier outcomes, preventing it from breaking its host isolation (e.g. by issuing rogue I/O requests to unauthorized areas on the disk).

With direct mediation, classifiers have direct write access to the request's command structure. They 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). Alternatively, classifiers can forward the request to several of NVMetro's predefined paths. 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 a 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.

For the sake of flexibility, NVMetro implements *iterative routing* where a request traverses multiple hops. To this end, we set up a state machine for each request; beyond request forwarding, classifiers can also set up *hooks* that will reinvoke them upon an event (e.g. fast path completion, return from UIF). 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.

C. Userspace I/O functions

UIFs attach to an NVMetro controller's notify path to exchange storage events with a classifier. They receive guest NVMe requests coming from the NSQ, do their tasks (encryption, compression, etc.), then return a completion entry via the NCQ. Each completion entry includes an *NVMe status code* to indicate the next step for the request, and an *auxiliary feedback* for exchanging additional data with the classifier.

We expose the notify queues (NSQs/NCQs) as file descriptors, which UIFs map into their address space to gain direct access to these queues. They can poll, send and receive data through these queues without using system calls; or they can block waiting for events using standard OS APIs (e.g. `epoll`). With these two features, UIFs can use the same *adaptive polling* approach used in our router to reduce its own CPU usage while maintaining low storage latency.

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 a **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 further examples of UIF implementations under our framework in Section IV.

IV. Use cases

In the following sections, we detail three examples of storage virtualization functions implemented with NVMetro: a function for managing copy-on-write disk snapshots with in-kernel translation caching; 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.

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 backend NVMe device's namespace or partition. We then use the control interfaces to insert eBPF classifiers and attach UIFs for the desired storage function.

A. Disk snapshotting

We implemented a storage function for managing virtual disk snapshots, with focus on a fast in-kernel cluster translation cache to improve I/O performance. For a high-level overview, we split a virtual drive into fixed-size *clusters*, which serves as the unit of snapshotting. Virtual disk clusters are managed by a *mapping table* managed by a UIF, where each mapping table entry translates a cluster index into physical locations on the NVMe drive. Figure 2 shows the I/O request lifecycle of our snapshotting function.

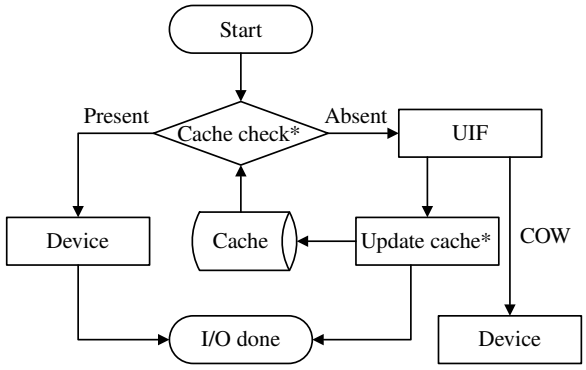

Fig. 2. Lifecycle of an I/O request with NVMetro snapshotting. The cache is stored inside the eBPF classifier. Asterisks (*) denote classifier invocation points. COW = copy-on-write.

I/O classifier: adjustable translation caching. A block address translation from virtual LBA to physical LBA involves multiple disk reads. Moreover, if we translate all disk accesses inside the UIF, each access will need to be redirected to the UIF, reducing storage function performance. However, it is equally undesirable to implement multiple virtual disk formats inside the kernel due to their complexity. To address this shortcoming, we decouple our function into two separate components, the UIF that handles disk format parsing and address translation and a format-agnostic translation cache inside an eBPF classifier, akin to translation lookaside buffers in CPUs.

We implemented a 4-way set-associative cache that transforms the index of each virtual cluster into its on-disk

physical location. Cache replacement is done using the PLRUm algorithm [15], which marks each cache line with most-recently-used (MRU) bit which is set on access. As the caching layer is based on eBPF, its parameters are completely adjustable. The size of the cache, its replacement policy and associativity can all be adapted at runtime. It can also be replaced with a simple non-caching classifier that redirects all requests to the UIF in high-security scenarios.

Figure 3 represents the properties of each cache line: a validity bit; a bit signaling whether the cluster is writable or read-only; a MRU bit to implement our replacement policy; the tag corresponding to its index; and finally its physical address.

Packed metadata word				
Valid	Writable	MRU	Tag	Physical LBA
Valid	Writable	MRU	Tag	Physical LBA
Valid	Writable	MRU	Tag	Physical LBA
Valid	Writable	MRU	Tag	Physical LBA

Fig. 3. Structure of a translation cache set.

Our translation cache stores its cache lines using eBPF map arrays. For efficiency purposes, we pack four 3-tuples (Valid, Writable, MRU) of each cache set into a single word. Each cache operation therefore only needs one read/write on said word instead of reading cache lines one by one.

On each request, the classifier checks whether it can be satisfied by the cache. A read match is indicated by the virtual cluster index's tag matching a valid cache line's tag; a write match additionally checks that line's writable bit. If the request is satisfied by the match, our I/O classifier rewrites the request's target address and directly forwards it to hardware. Failing that, the request is forwarded to the UIF for further processing.

Userspace I/O function. Our UIF does double duty: managing virtual disk address translation and snapshotting using on-disk mapping tables, and keeping the classifier translation cache up to date. For its first duty, the UIF makes use of a two-level mapping table structure, which is protected by NVMetro and transparent to the VM. We implement copy-on-write snapshotting by duplicating mapping tables and then marking their entries as read-only. Writes onto read-only clusters are handled by copying the cluster onto a different location followed by updating the mapping table correspondingly. The cluster is duplicated asynchronously using `io_uring`.

Finally, to update the classifier cache, the UIF uses the auxiliary feedback mechanism to send the corresponding cache line's Valid, Writable and Physical LBA properties, along with a request disposition of *Keep Cache* (to preserve the existing translation entry) and/or *Forward* (to forward the request to hardware if necessary).

Optimizations. We present below several optimizations implemented in our snapshotting function to avoid performance issues caused by virtual disk address translation.

- **Cluster I/O alignment.** For translation efficiency, virtual disks are organized into clusters composed of multiple physical disk blocks. This feature is common across virtual disks (QEMU QCOW, SPDK logical volumes, LVM) and file systems. For example, QCOW uses a default cluster size of 64 KB for its disk images. In normal situations, a virtual I/O request is translated into one corresponding physical read or write, with the I/O offset adjusted according to the translation table. However, when an I/O request crosses cluster boundaries, one virtual request could be translated into multiple physical requests, as the disk block range covered by the request is no longer linear. Figure 4 shows an illustration of this effect. With our snapshotting function, a misaligned I/O request also prevents the classifier from using cached cluster addresses due to the I/O splitting requirements. To avoid such situations, we set the *Namespace Optimal I/O Boundary* identification field on our virtual NVMe namespace. This field instructs the guest to split I/O requests on cluster boundaries on its own, so that I/O operations would use our translation cache optimally.
- **Read-modify-write and write amplification.** On copy-on-write virtual disks, a write onto a read-only cluster (e.g. one shared with another snapshot) must be redirected to a copy of that cluster. If the write does not cover the entire cluster, the storage function must read the entire original cluster; update the content with the written data; then write the entire content to another location. Figure 5 illustrates two consequences of this scenario: an *extra read* from having to recover the original content, and *write amplification*, as regardless of how small a virtual write is, it must be translated to a full physical cluster write. These issues cause a performance impact on small writes which worsens with increasing cluster sizes. To mitigate this issue, our storage function specifies the *Namespace Preferred Write Granularity* and *Namespace Preferred Write Alignment* identification fields to encourage writes to completely fill a cluster when possible.

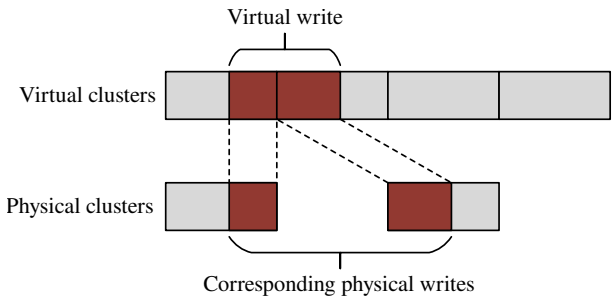

Fig. 4. Effect of cluster I/O misalignment.

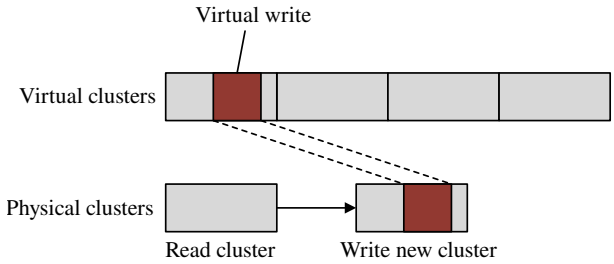

Fig. 5. Effect of read-modify-write and write amplification.

B. 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 6 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 a UIF, which decrypts and encrypts data during reads and writes respectively. This function aims to demonstrate the mediation options of an NVMetro classifier, which we describe below.

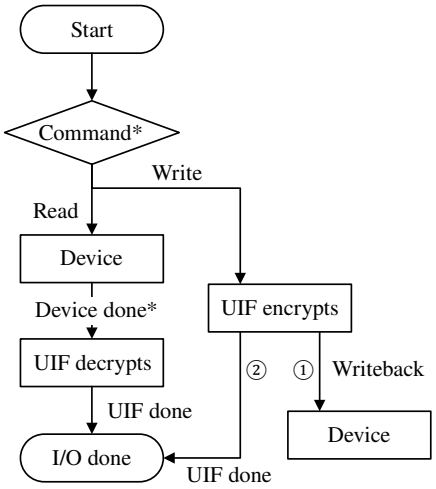

Fig. 6. Lifecycle of an I/O request with data encryption. Asterisks (*) denote classifier invocation points.

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 6). 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 demonstrated both types of request modification available to a classifier:

- Direct mediation: by returning an 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.

C. 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.

In our I/O request pipeline, 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 writes 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.

D. Implementation effort

TABLE I
CODE SIZES OF NVMETRO STORAGE FUNCTIONS.

Function	Component	Lines of code
Snapshot	Classifier	217
Snapshot	UIF	1634
Encryptor	Classifier	32
Encryptor	Normal UIF	520
Encryptor	SGX UIF + enclave	501
Replicator	Classifier	16
Replicator	UIF	307
Framework	—	1116

Table I shows a breakdown of the lines of code needed for each of our storage functions. The code size statistics above demonstrate the ease in implementing fast and simple storage functions with NVMetro. Firstly, our architecture is highly *pluggable*, naturally separating the fast path and slow business logic. Developers can experiment with combining different pairs of classifier/UIF without needing an explicit redesign or significant modifications. In our disk encryption example, we shared the same classifier and 80% of UIF code between our normal and SGX encryptor functions. Similarly, in our snapshotting example, our caching layer is modularized; users can enable and disable caching by swapping the classifier without rebooting or recompiling the code, while our UIF transparently handles any cache misses.

Secondly, storage developers can easily and incrementally upgrade an NVMetro function with less effort. As an example, our disk replication function was originally a synchronous implementation, using familiar I/O API such as `pread` and `pwrite`. Using our UIF framework, which contained built-in `io_uring` support and abstracted over both sync and async programming models, we were able to keep 66% of the code (request handler structures, queue configuration and main loop) while switching the entire function to async I/O.

E. Security properties of NVMetro

NVMetro implements multiple layers of isolation to protect the host and unrelated VMs from malicious UIF and eBPF classifiers. This is the key component of NVMetro isolation that prevents UIFs and classifiers from making unauthorized actions. Firstly, both components run under unprivileged contexts (user processes and eBPF sandboxes, respectively). This sandboxing is mandatory and cannot be disabled, since storage functions running in a privileged context may damage the host and/or unrelated data. We execute classifiers in a restricted eBPF-based sandbox, having only access to the current NVMe request command block. To cleanly separate the fast path and in-UIF business logic, we restrict the classifier's communications, which can only be done through NVMetro's request router. UIFs run as normal usermode programs, and NVMetro does not give it any special privileges beyond access to the notify path, which is also under the request router's control.

Additionally, NVMetro sandboxes every I/O request before it reaches the physical device. Our sandboxing protects other NVMe namespaces and data partitions from malicious UIFs and classifiers that aim to steal or destroy data belonging to the host or other VMs. To do so, by default, NVMetro forbids access to the admin queues, dangerous commands (e.g. queue controls, security, format NVM). We implemented sysfs attributes to let administrators bypass this restriction as needed. NVMetro also checks that submitted commands are limited to a single NVMe namespace or partition. Namely, in our use cases presented above, UIFs only have access to VM disk partitions they need, not the entire NVMe device.

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 UIFs presented in Section IV with multiple different workloads: firstly, benchmarks of I/O performance under various configurations with `fio` [16]; and secondly, database evaluations using the YCSB suite [17].

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 Linux 5.15.

`fio` evaluation setup. To evaluate NVMetro's 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.

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

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$^{\text{th}}$-percentile latencies for each configuration.

YCSB evaluation setup. We used 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 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 [18]); 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.

Figure 7 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 $2.7\times$ 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 8 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 $3.4\times$ higher median random read latency and $4.1\times$ 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.

Figure 9 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 10 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. Snapshotting evaluations

We evaluate NVMetro's snapshotting function in comparison to several other implementations: QEMU's QCOW image files (backed by the Ext4 filesystem) and `virtio-blk`; Linux

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

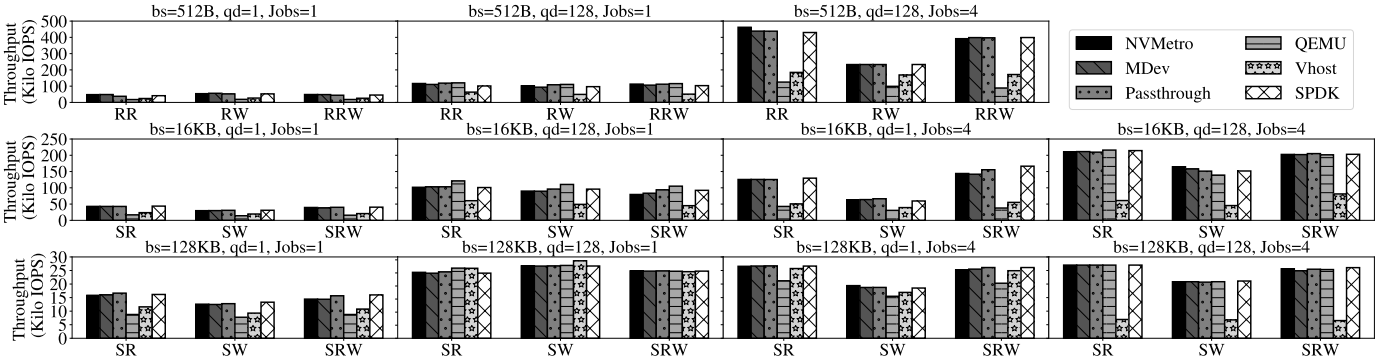

Fig. 7. Basic evaluations: `fio` performance for each workload configuration and storage virtualization method.

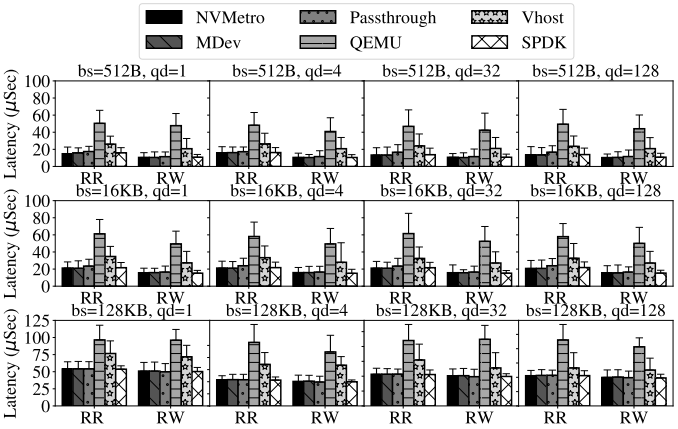

Fig. 8. NVMetro latency evaluation results. Columns denote median latency; 99th-percentile latency is shown in whiskers.

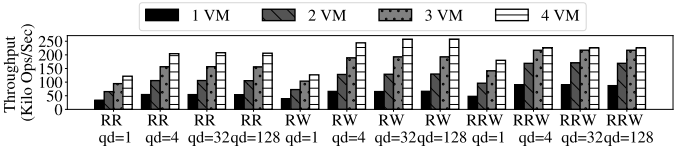

Fig. 9. NVMetro scalability evaluation results.

LVM and `vhost-scsi`; and finally, SPDK's logical volume and `virtio-blk`. With each solution, we created a 100 GB virtual disk with a fixed cluster size of 64 KB, fill 50% of the clusters with random data, then create a snapshot on top of the randomized disk content. Subsequently, to avoid reinitializing the virtual disk every `fio` execution, we run a `fio` benchmark script that cycles over various benchmark configurations while randomly accessing the drive with various block sizes, queue depths and number of parallel jobs (refer to Table II). At the end of the benchmark script, our virtual disk is finally reinitialized from snapshot, and the written data discarded using appropriate unmap/TRIM commands available to each storage solution.

Figure 11 shows the I/O throughput of each solution for each part of the benchmark script. We observe that NVMetro performs comparably or better than QEMU, LVM and SPDK in virtually all benchmark configurations; most notably at 512B RW/QD1/1 job where its I/O rate is 3.8× higher than other solutions on average. Similarly, at 128KB RW/QD1/1 job,

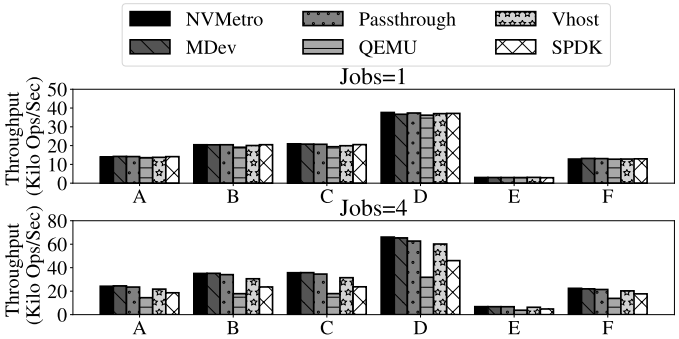

Fig. 10. YCSB throughput for each workload type (A-F).

NVMetro is 39%/90% faster than the next best contenders LVM and SPDK. However, NVMetro performs worse than SPDK by a margin of 14% and 10% at 512B RR/QD1/1 job and 16KB RW/QD1/1 job respectively.

At 512B block sizes, QEMU and LVM perform poorly compared to NVMetro and SPDK, with LVM being 2.3×–3.8× slower than NVMetro at 512B RW-RR/QD128/4 jobs respectively. However, at 128 KB block size, LVM reaches comparable performance to NVMetro, while QEMU and SPDK are approximately 2× slower at this block size.

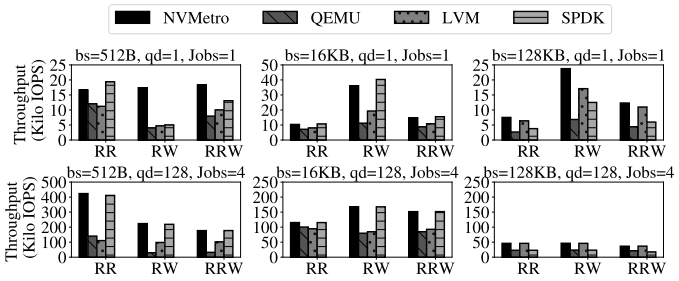

Fig. 11. Disk snapshotting evaluations with `fio`.

D. 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

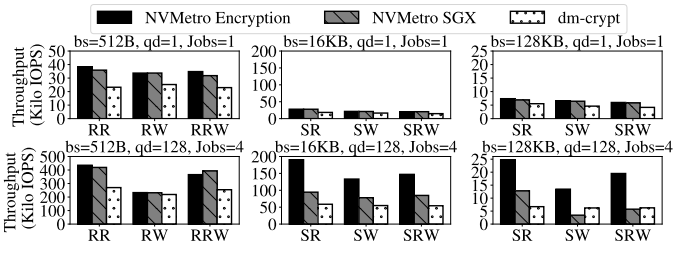

Fig. 12. Disk encryption evaluations with `fio`.

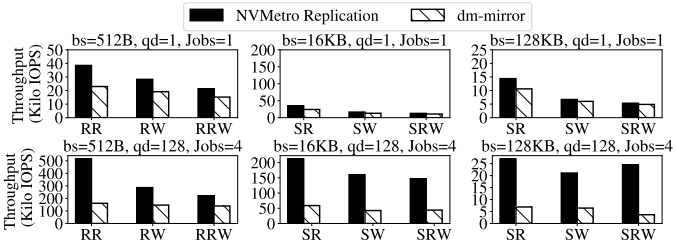

Fig. 13. Disk replication evaluations with `fio`.

scenarios presented above. Our non-SGX UIF uses 2 threads; our SGX UIF uses 1 worker + 1 SGX switchless thread.

Overall, Figure 12 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).

E. 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 13), 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.

F. 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 14). Device passthrough predictably performs the best among all tested configurations. MDev-NVMe, NVMetro and QEMU perform similarly, using ≈ 85% more total CPU than passthrough at 512B/QD1/1 job, and ≈ 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, SPDK uses the most CPU time, with a ≈ 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.

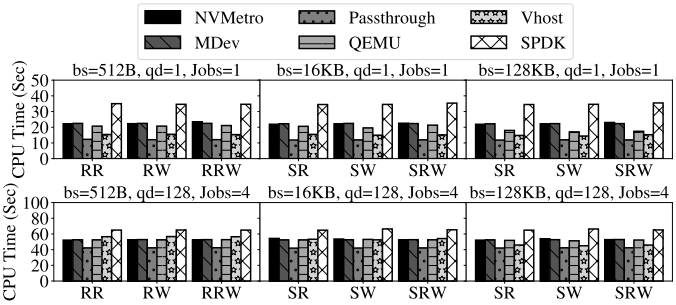

Fig. 14. CPU consumption of `fio` with basic evaluation.

Disk encryption (Figure 15). At (512B, 16K, 128K) QD1/1 job, our encryption UIF uses around 2.7×, 2.4× and 2.1× 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 ≈ 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.

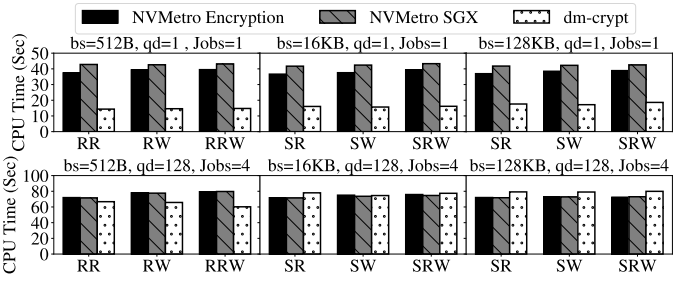

Fig. 15. CPU consumption of `fio` with disk encryption.

Snapshotting (Figure 16). Among all configurations, NVMetro has a similar CPU load to SPDK, with the exception of 128KB/QD128/4 jobs where NVMetro uses 27% and 41% more CPU than SPDK at random reads and writes (while being 2× faster in return). At lower queue depths, QEMU and LVM have lower CPU usage than polling-based solutions; however, at higher I/O rates such as (512B, 16KB)/QD128/4 jobs, their CPU consumption increases to match NVMetro and SPDK.

Disk replication (Figure 17). At 512B/QD1/1 job, 512B/QD128/4 jobs and 128K/QD128/4 jobs, NVMetro incurs a

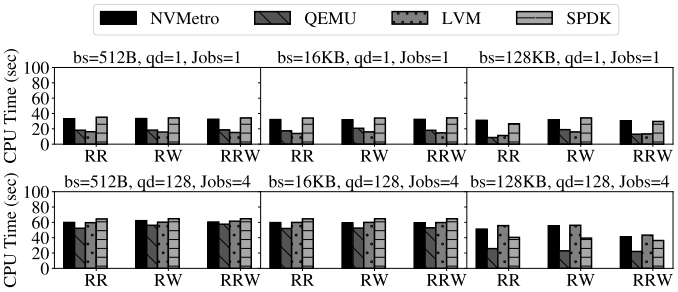

Fig. 16. CPU consumption of `fio` with snapshotting.

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.

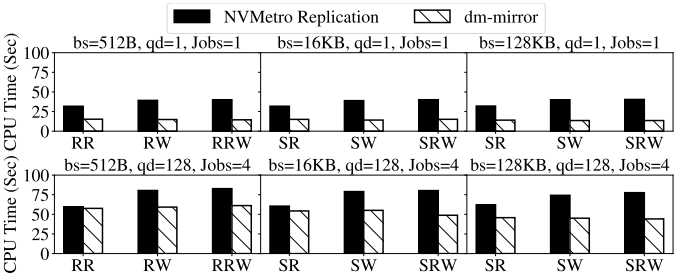

Fig. 17. CPU consumption of `fio` with disk replication.

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

As we claimed in Section I, NVMetro's storage framework is more flexible and easier to use than existing systems. In this section, we support our claims by contrasting our storage function implementations 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 [19]). 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, NVMetro decouples the eBPF classifier from the UIF containing business logic (encryption code, disk format processing, etc.) Our design brings several benefits over in-kernel processing. Firstly, it increases the storage function's security and reliability through stronger sandboxing. The eBPF classifier is restricted to a small API for controlling an I/O request's lifetime, while the main work is delegated to an unprivileged UIF which can be restarted or updated as needed. Secondly, NVMetro gives control over the storage function's resource use. For example, multiple VMs can share the same UIF, reducing the use of costly I/O polling threads, or they can use separate UIFs to increase security. Thirdly, storage functions can integrate any new technologies or libraries without losing I/O performance or having to deal with the difficult programming environment of an OS kernel. eBPF classifiers can also be reused for different functions, speeding up development and reducing bugs.

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 a new 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 MDev-NVMe. 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. More hypervisor operations are then needed to signal the VM (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, as UIFs simply communicate with the kernel using standard POSIX APIs.

VI. Related works

General computational storage architecture. SNIA's Computational Storage Architecture and Programming Model [20] defines a general structure of computational storage applications, where different storage engines (e.g. eBPF-based) can be embedded into various device classes. It also defines several computational storage function types for these engines.

Virtual storage providers. SPDK [21] 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, including the Vhost service for providing virtual disks to VMs. SPDK Vhost-NVMe [22] builds a virtual NVMe interface on top of SPDK with an optimized I/O path. NVMe Direct [23] introduces an API similar to the RDMA Connection Manager to grant userspace programs direct access to NVMe's high-performance queues. NVMe Direct 2.0 [24] is a preloadable shared library that shadows I/O functions (similar to libraries such as `libsdp` and `libvma`) to improve performance without changing application code.

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 [25] or an external process (e.g. SPDK) via `vhost-user` [22], [26]. MDev-NVMe [1] describes an 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.

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. Griffin [27] envisions an API set using eBPF to add logic to storage apps running on edge computing nodes. Kourtis et al. [28] follow in the same line by running eBPF on top of NBD, and propose ways to use eBPF for KV store and SQL offloading. Huang and Paradies [29] compare eBPF and WebAssembly's various aspects (safety, compatibility, performance, etc.), and show how to developing these technologies for storage 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.

Most similarly to NVMetro, XRP [2] inserts BPF hooks into Linux's NVMe driver submission path to accelerate filesystem translation. Compared to XRP, NVMetro provides a virtualization-specific design that maintains compatibility with the NVMe protocol, plus a framework for UIFs with a specific, high-performance communication channel (versus the `bpf()` syscall in the case of XRP).

Hardware-based solutions. In this category, LeapIO [30] presents a new storage stack that offloads virtualization tasks onto on-disk processors coupled with smart memory and NIC sharing to improve performance. FastPath [31] adds a FPGA-based computing engine between the host and storage device, while exposing an API to offer a fast path to applications needing high I/O performance. FastPath_MP [32] extends FastPath with multiple I/O queues to take advantage of the parallelism offered by NVMe devices. Similarly, FVM [33] interposes NVMe devices with FPGA to virtualize storage using I/O queue emulation and storage address translation. A current trend also aims to integrate storage functions directly into hardware through specialized components. Studies such as [34]–[36] propose embedding functionalities at the hardware component level, facilitating, for instance, VM migration via SmartNICs ([36]) or using FPGAs to accelerate storage access for HPC applications [34]. All these works are orthogonal to NVMetro and can use our solution as a software implementation of their hardware components, as NVMetro introduces flexibility in the implementation of storage functions and multiple data paths.

TABLE III
COMPARING NVMETRO TO EXISTING SOLUTIONS

	Flexibility	Performance	Hardware	CPU	Intrusiveness
NVMetro	✓	✓	✓	✗	✓
SPDK	✗	✓	✓	✗	✗
LeapIO	✓	✓	✗	✓	✓
Vhost	✗	✗	✓	✓	✓
MDev	✗	✓	✓	✗	✓
QEMU	✗	✗	✓	✓	✓

Comparison with existing works. In the previous sub-sections, we presented various existing solutions for implementing network functions. In this section, we will compare NVMetro with these solutions. The main advantage that NVMetro offers is the flexibility it introduces, both in implementing new network functions and in the possibility of having multiple data paths. Table III provides a comparison of NVMetro with the existing solutions. The criteria we used are: flexibility (ease of implementation of complex storage functions), performance, hardware (the necessity of a new hardware component), CPU load, and intrusiveness (the need to modify applications for each solution). While NVMetro's primary limitation is its impact on CPU load, its strong performance and flexibility largely compensate this limitation. In contrast, hardware-based solutions like LeapIO offer good performance but require specialized hardware and deployment processes.

VII. CONCLUSION

We introduced NVMetro, our storage framework that provides virtual machines with fast and flexible storage services. NVMetro uses a central I/O router with eBPF and active polling to route I/O requests for implementing complex storage functions. We detailed our implementation of three storage functions to show our framework's flexibility and ease of use. Our implementations demonstrate the ease of adding custom storage logic to the host kernel without compromising security. We evaluated these functions under numerous configurations to show NVMetro's strong performance and scalability.

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] B. Peng, J. Yao, Y. Dong, and H. Guan, "Mdev-nvme: Mediated pass-through nvme virtualization solution with adaptive polling," *IEEE Transactions on Computers*, vol. 71, no. 2, pp. 251–265, 2022.

[2] Y. Zhong, H. Li, Y. J. Wu, I. Zarkadas, J. Tao, E. Mesterhazy, M. Makris, J. Yang, A. Tai, R. Stutsman, and A. Cidon, "XRP: In-Kernel storage functions with eBPF," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, Jul. 2022, pp. 375–393. [Online]. Available: https://www.usenix.org/conference/osdi22/presentation/zhong

[3] 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.

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

[7] National Vulnerability Database, "CVE-2015-4036," https://nvd.nist.gov/vuln/detail/CVE-2015-4036, 2015.

[8] ——, "CVE-2019-14835," https://nvd.nist.gov/vuln/detail/CVE-2019-14835, 2019.

[9] ——, "CVE-2024-0340," https://nvd.nist.gov/vuln/detail/CVE-2024-0340, 2024.

[10] T. Dinh Ngoc, B. Teabe, G. Da Costa, and D. Hagimont, "Flexible nvme request routing for virtual machines," in *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2024.

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

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

[13] ——, "Linux socket filtering aka Berkeley packet filter (BPF)," https://www.kernel.org/doc/html/latest/networking/filter.html, 2021.

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

[15] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, "Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite," in *Proceedings of the 42nd Annual Southeast Regional Conference*, ser. ACM-SE 42. New York, NY, USA: Association for Computing Machinery, 2004, p. 267–272.

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

[17] 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.

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

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

[20] 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.

[21] 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.

[22] 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.

[23] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "NVMeDirect: A user-space I/O framework for application-specific optimization on NVMe SSDs," in *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[24] J.-S. K. H.-J. Kim and J.-S. Kim, "NVMeDirect 2.0: An enhanced user-space I/O framework for NVMe SSDs," *Flash Memory Summit*, 2017.

[25] 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.

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

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

[28] 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.

[29] W. Huang and M. Paradies, "An evaluation of WebAssembly and eBPF as offloading mechanisms in the context of computational storage," *arXiv preprint arXiv:2111.01947*, 2021.

[30] 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.

[31] 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.

[32] ——, "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.

[33] 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.

[34] R. Wertenbroek, Y. Thoma, and A. Dassatti, "A portable linux-based firmware for nvme computational storage devices," *ACM Trans. Storage*, Sep. 2024, just Accepted. [Online]. Available: https://doi.org/10.1145/3697352

[35] Y. Zou, A. Awad, and M. Lin, "Directnvm: Hardware-accelerated nvme ssds for high-performance embedded computing," *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 1, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3463911

[36] J. Zhao, R. Shu, L. Qu, Z. Yang, N. E. Jerger, D. Chiou, P. Cheng, and Y. Xiong, "Smartnic-enabled live migration for storage-optimized vms," ser. APSys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 45–52. [Online]. Available: https://doi.org/10.1145/3678015.3680487

Tu Dinh Ngoc is a researcher at the SEPIA research group at IRIT Toulouse, France. His main research interests are in operating systems, virtualization and storage techniques.

Boris Teabe is an Assistant Professor at INP Toulouse, France. His main research interests are in scheduling in virtualization, cloud computing and operating systems.

Georges Da Costa is a Professor at the Université Toulouse III - Paul Sabatier. His focus is on the energy efficiency of HPC and large scale infrastructures (clusters, grids, clouds).

Daniel Hagimont is a Professor at INP Toulouse, France and a member of the IRIT laboratory, where he works on operating systems, distributed systems and middleware research.