Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure

Shuai Xue, Shang Zhao, and Quan Chen, *Shanghai Jiao Tong University and Alibaba Cloud;* Gang Deng, Zheng Liu, Jie Zhang, Zhuo Song, Tao Ma, Yong Yang, Yanbo Zhou, Keqiang Niu, and Sijie Sun, *Alibaba Cloud;* Minyi Guo, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/atc20/presentation/xue

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Open access to the Proceedings of the 2020 USENIX Annual Technical Conference is sponsored by USENIX.

Spool: Reliable Virtualized NVMe Storage Pool in Public Cloud Infrastructure

[†‡]Shuai Xue, [†‡]Shang Zhao, [†‡]Quan Chen, [‡]Gang Deng, [‡]Zheng Liu, [‡]Jie Zhang, [‡]Zhuo Song
[‡]Tao Ma, [‡]Yong Yang, [‡]Yanbo Zhou, [‡]Keqiang Niu, [‡]Sijie Sun, [†]Minyi Guo
[†]*Department of Computer Science and Engineering, Shanghai Jiao Tong University*
[‡]*Alibaba Cloud*

Abstract

Ensuring high reliability and availability of virtualized NVMe storage systems is crucial for large-scale clouds. However, previous I/O virtualization systems only focus on improving I/O performance and ignore the above challenges. To this end, we propose Spool, a reliable NVMe virtualization system. Spool has three key advantages: (1) It diagnoses the device failure type and only replaces the NVMe devices with actual media errors. Other data link errors are handled through resetting the device controller, minimizing data loss due to unnecessary device replacement. (2) It ensures the consistency and correctness of the data when resetting the controller and upgrading the storage virtualization system. (3) It greatly reduces the restart time of the NVMe virtualization system. The quick restart eliminates complaints from tenants due to denial-of-service during a system upgrade and failure recovery. Our evaluation shows that Spool provides reliable storage services with performance loss smaller than 3%, and it reduces restart time by 91% when compared with SPDK.

1 Introduction

In large-scale public clouds, the cores and memory are virtualized and shared by multiple tenants. A single physical server can serve up to 100 virtual machines (VMs) from either the same or different tenants [41]. On the physical server, VMs are managed with VM hypervisors, such as VMware [13], KVM [25], and Xen [14]. The hypervisors are also responsible for handling the interactions between the guest operating system in the VMs and the host operating system on the physical server.

Virtualizing I/O devices so that tenants can share them has attracted the attention of both industry and academia [15, 23, 31, 33, 40, 42]. A guest VM mainly stores and accesses its data on local devices through the I/O virtualization service with high throughput and low latency. For instance, the Big Three of cloud computing (Amazon EC2 I3 series [2], Azure Lsv2 series [3], and Alibaba ECS I2 series [1]) are providing the next generation of storage optimized instances for workloads that require high I/O throughput and low latency. These products are driven by local devices that eliminate the long latency over the network [8]. At the same time, accessing data from local devices increases the risk of a single point of failure as the reliability of data is dependent on the reliability of the host node.

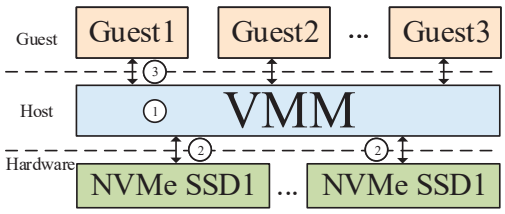


Figure 1: Virtualizing NVMe-based storage system.

Solid-state drives (SSDs) are often adopted as storage devices due to their high throughput and low latency compared to those of hard drives. In particular, the recent NVM Express (NVMe) interface [9] further increases the I/O performance of SSDs compared with the traditional SATA interface. Mainstream storage virtualization solutions, such as Virtio [29], support NVMe devices. Because serious performance degradation is observed in I/O virtualization [22], userspace NVMe driver in QEMU [43], Storage Performance Development Kit (SPDK) [38], SPDK vhost-NVMe [39], and Mdev [27] have been proposed to further improve the I/O throughput of virtualized NVMe devices.

While prior researchers focused on improving the read/write throughput and reducing the latency of virtualized NVMe devices, they ignored the reliability problem although it is equally important. In large-scale public clouds, NVMe device failures occur due to heavy use and the need for NVMe virtualization systems to be upgraded often to add new features or apply new security patches. Emerging NVMe virtualization systems fail to handle failure recovery and system upgrades efficiently. To better explain this problem, Figure 1 shows an example where multiple tenants share a virtualized NVMe storage system on a physical node.

With emerging virtualized storage systems, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through either cold-plug or hot-plug. This failure recovery mechanism results in unnecessary data loss from the failed NVMe device. The statistics of our in-production cloud show that only 6% of 300,000 device failures involve media errors that can only be resolved by replacing with a new device. Other device failures are caused by data link errors that can be resolved by resetting the NVMe device controller. Resetting an NVMe device's controller would not result in data loss from the device, and we can perform the reset operation fast without removing the failed device (② in Figure 1) and restarting the virtualization system (① in Figure 1).

The standard procedure for upgrading the virtualized storage system on a node is stopping the daemon process that runs the system, updating the binary file, and then initializing the whole software stack of the virtualization system again [17]. In this period, all the I/O devices on the node are inaccessible due to the lack of an I/O virtualization system. Our measurement shows that the software initialization procedure already spends approximately 2.5 s probing all the I/O devices and SPDK's Environment Abstraction Layer (EAL) [38] (to be discussed in detail in Section 5). This long downtime hurts the user experience. A possible solution to reduce the impact of the upgrade is migrating the VMs (and the corresponding data) to other nodes [19, 41]. However, live VM migration is too costly for regular backend updates, especially when a large amount of backend requires updating, for example, when applying an urgent security patch.

However, the data written by the Guest VMs may be lost when resetting the controller or performing the upgrade (③ in Figure 1). The loss happens in the case that the data persisted in the NVMe device (still in the submit queue) when the reset operation or the process restart was performed. No prior work on NVMe virtualization has considered such a reliability problem.

To resolve the above problems, we propose *Spool*, a holistic reliable virtualized NVMe storage system for public clouds with local disks. Compared with prior NVMe virtualization systems, Spool has the following key advantages: (1) It diagnoses the device failure type and only replaces the NVMe devices with actual media errors. Other data link errors are handled through resetting the controller, minimizing data loss due to the unnecessary disk replacement. (2) It ensures the consistency and correctness of the data when resetting the controller and upgrading the virtualization system. (3) It greatly reduces the restart time of the NVMe virtualization system to approximately 100 milliseconds. The quick restart eliminates complaints from tenants due to denial-of-service during system upgrades and failure recovery.

To be more specific, Spool is comprised of a *cross-process journal for recovery*, an *isolation-based failure recovery component*, and a *fast restart component*. The cross-process journal resides in the shared memory and records the data status from all the VMs. Even if Spool is restarted, the data in the journal is accessible for the new Spool process. Furthermore, an instruction merge is proposed to eliminate the inconsistency of the journal with minimal overhead. An "instruction" is a step within a transaction that updates the journal. The failure recovery component diagnoses the NVMe device error. Based on the error code, Spool either isolates and replaces the devices that have media errors or resets the controller (using the journal for reliability). The restart component records the runtime data structures of the current Spool process in the cross-process journal. By reusing the data structures at the restart for a system upgrade, we significantly reduce the downtime.

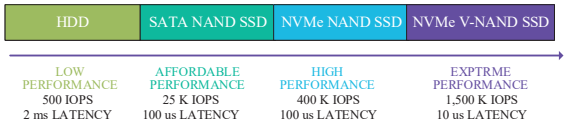


Figure 2: Development of the hardware I/O performance.

To the best of our knowledge, Spool is the first holistic virtualized system that is capable of handling hardware failure and NVMe virtualization system upgrades reliably. Spool is currently deployed in an in-production cloud that includes more than 20,000 physical nodes and 200,000 NVMe-based SSDs.

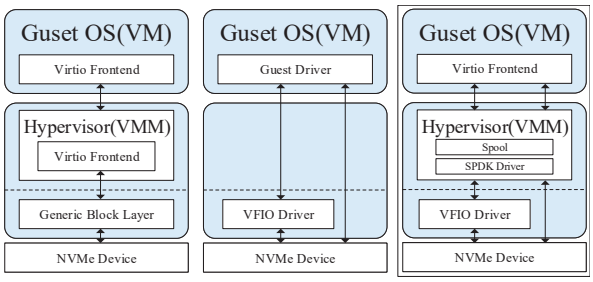
The main contributions of this paper are as follows.

- **An instruction merge-based reliability mechanism.** The instruction merge eliminates data inconsistent with the cross-process journal for recovery even if abnormal exits occur.
- **A restart optimization method.** The method greatly reduces the downtime of Spool during the upgrade and enables frequent system upgrades for adding new features and applying patches without affecting the tenants.
- **A hardware fault processing mechanism.** The mechanism diagnoses the device failure types and only replaces the NVMe devices with media errors, minimizing data loss due to unnecessary disk replacement.

Our experimental results show that Spool provides reliable storage services based on a shared memory journal with less than 3% performance loss, and it reduces the system restart time by 91% when compared to SPDK.

2 Background and Motivation

In this section, we introduce the virtualized NVMe storage systems and the motivation behind the design of Spool.



(a) Virtio (b) Passthrough based on VFIO (c) Spool based on SPDK

Figure 3: Comparison of NVMe virtualization mechanisms.

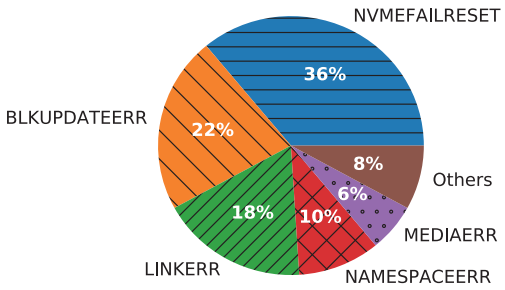


Figure 4: Breakdown of NVMe hardware failures.

2.1 Virtualized NVMe Storage Systems

The performance of an I/O device is impacted by both the storage media and the I/O software stack. As shown in Figure 2, Samsung NVMe SSD devices based on the latest V-NAND technology have increased the IOPS to 1.5 million and reduced the latency to 10 microseconds. In this scenario, the traditional SATA (Serial ATA) [34] interface for storage devices has become the performance bottleneck for such SSDs. Due to the limitation of the Advanced Host Controller Interface (AHCI) architectural design, the theoretical data transmission speed of the SATA interface is only 600 MB/s [34]. To solve the I/O bottleneck brought by the interface, the NVMe (Non-Volatile Memory Express) protocol [9] is designed and developed using a PCIe interface instead of SATA. Currently, NVMe supports deep queues with up to 64K commands to devices within a single I/O queue [9].

In public clouds, instead of selling raw hardware infrastructure, cloud vendors typically offer virtualized infrastructure as a service to maximize hardware resource utilization [16, 18]. Virtualization technology has shown its heroism, especially in the birth of hardware virtualization technology, such as Intel VT technology, which has greatly expanded the application scope of virtualization technology. There are three parts to the realization of virtualization: *CPU virtualization, memory virtualization*, and *I/O virtualization*. Among them, I/O virtualization requires more focus, and its performance directly determines the performance of the guest VM [22, 30, 32].

There are generally three I/O virtualization mechanisms: Virtio [29], VFIO [36], and SPDK-based userspace applications [38]. Figure 3 shows a comparison between Virtio, VFIO, and our SPDK-based design, Spool.

As for Virtio, the frontend exists in a guest OS, while the backend is implemented in a hypervisor, such as QEMU [12]. The frontend transfers I/O requests to the backend through the virtqueue, implemented as ring buffers, including *available ring* and *used ring* buffers. Available ring buffers could save multiple I/O requests driven by the frontend and transfer them to the backend for batch processing, which can improve the efficiency of information exchange between the client and hypervisor. However, a problem remains that each I/O request passes through the I/O stack twice for guest and host, whereas in modern storage devices based on NAND flash, the throughput and latency of VMs can only achieve 50% of the native performance [27].

As for VFIO, VMs directly access an NVMe device through passthrough, relying on hardware support (e.g., Intel VT-d). A VM approaches near-native performance on both latency and throughput with passthrough. However, a single device can only be assigned to one guest client. On the contrary, a host often runs multiple clients in a virtualized environment. It is difficult to ensure that each client can get a directly assigned device. Also, a large number of devices are allocated to clients independently, increasing the number of hardware devices as well as the cost of the hardware investment.

The Storage Performance Development Kit (SPDK) provides a set of tools and libraries for writing high-performance, scalable, user-mode storage applications. The bedrock of SPDK is a userspace, polled-mode, asynchronous, lockless NVMe driver [38]. SPDK enables zero-copy, highly parallel access direct to SSDs from a userspace application. User-mode drivers help improve the stability of the host operating system because they can only access the address space of the processes running them, and a buggy implementation does not cause system-wide problems. Spool is proposed based on the SPDK NVMe driver but focuses on the reliability of the virtualized storage system.

2.2 Reliability Problems

All the above I/O virtualization mechanisms ignore the high availability and reliability problems, although they are equally important in public clouds. To be more specific, state-of-the-art SPDK-based applications result in *unnecessary data loss* and *poor availability* when dealing with failed NVMe devices and upgrading applications, respectively.

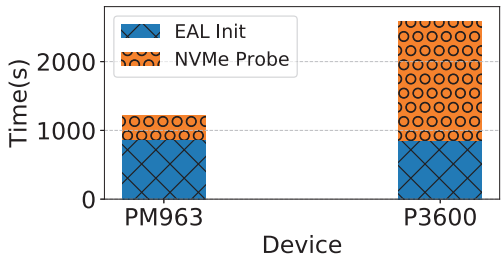


Figure 5: Breakdown of SPDK's start time on two NVMe SSDs.

2.2.1 Unnecessary Data Loss

If an NVMe device failure is detected on the hardware node, the device is in the failed state. When a device failure occurs on a node, all the VMs on the node are de-allocated and migrated to a healthy node by a standard procedure [11, 41]. After that, all the data on the failing node are securely erased. The victim tenants' data are lost and the tenants must proactively load their data on the new node again. With emerging virtualized storage systems like SPDK, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through hot-plug.

The above method results in significant unnecessary data loss because a single NVMe device may store data from multiple tenants, and NVMe devices have higher storage density, more vulnerable components (e.g., a Flash Translation Layer), and relatively higher failure rates. To demonstrate this problem in detail, we collected 300,000 NVMe device failures in our in-production environment. Figure 4 shows the breakdown of device failures. Most of the failures, 36%, are due to the NVMe controller failure error (NVMEFAILRESET). BLKUPDATEERR is the block update error. LINKERR is the PCIe interconnect link error. NAMESPACEERR is the NVMe device's namespace error. The pie chart shows that only 6% of the hardware failures are due to real media errors (MEDIAERR). Our investigation shows that most failures are caused by errors in the data link layer (e.g., namespace error, hardware link error, NVMe reset fail error), and these failures can be resolved by simply resetting the NVMe controller.

In summary, the current failure recovery method with SPDK results in significant unnecessary data loss.

2.2.2 Poor Availability

I/O virtualization systems tend to be upgraded frequently to add new features or apply security patches. When upgrading an I/O virtualization system, the key requirement is minimizing the I/O service downtime while ensuring the correctness of the data. There are two methods available to cloud vendors: VM live migration and live upgrade. Unfortunately, VM live migration is too costly for regular backend updates, especially when a large amount of backend requires updating, for example, when applying an urgent security patch, and for storing optimized instances with local NVMe storage , VM live migration is not even supported by cloud vendors [11]. The only way for us is to support the live upgrade and eliminate the downtime as much as possible.

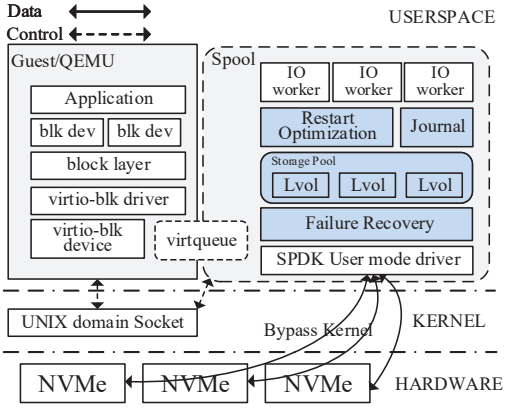


Figure 6: Design of Spool.

The I/O virtualization system must be restarted to complete the upgrade. With SPDK, we need to initialize the DPDK EAL library, probe the NVMe devices, and initialize the internal data structure of SPDK itself. SPDK spends a different amount of time in the "probe devices" step when resetting the controllers of different devices. The time required for each step is shown in Figure 5. As can be observed from this figure, the service downtime caused by the live upgrade is up to 1,200 ms for Samsung PM963 SSD. For Intel P3600, the total service downtime will be longer and lasts up to 2,500 ms.

In summary, the long downtime hurts the availability of the I/O virtualization system.

2.3 Design Principle of Spool

To resolve the unnecessary data loss and poor availability problems, we propose **Spool**, a holistic NVMe virtualization system. Spool is designed based on three principles:

- It should be able to identify the causes of device failure and adopt different methods to handle each failure. In this way, Spool eliminates most unnecessary NVMe device replacement.
- It should be able to optimize the restart procedure during a live upgrade so that the downtime can be minimized.
- It should be able to ensure that data access requests from the guest OS are not lost during a controller reset and live system upgrade.

3 Methodology of SPOOL

Figure 6 shows the design architecture of Spool, where the blue components are new relative to SPDK. Based on Spool,

the NVMe devices on a node are virtualized and organized into a *Storage Pool (hence, "Spool")*. The virtualized NVMe devices are divided into multiple logical volumes that are managed through the buddy system [28]. The logical volumes are exposed to the guest OS in the form of block devices. As shown in the figure, the guest drivers communicate with Spool over shared memory. Specifically, the I/O worker on the host node polls I/O requests from the vhost virtqueue of block devices and submits to the corresponding physical devices. Spool is comprised of a *cross-process journal*, an *isolation-based failure recovery component*, and a *fast restart component*. Based on the three components, Spool ensures high reliability and availability of the storage pool.

The cross-process journal records each I/O request and its status to avoid data loss. The journal provides data access across process lifecycles, even if Spool restarts for an upgrade or exits abnormally. An instruction merge mechanism is proposed to eliminate the possible inconsistency of the journal itself due to an abnormal exit and to avoid the copy overhead of atomic operations.

The restart component records the runtime data structures of the current Spool process in shared memory. Spool catches the termination signals including *SIGTERM* and *SIGINT* to ensure the completion of all INFLIGHT I/O requests before actual exit. Spool reuses the data structures at the restart, thus significantly reducing the downtime spent on initializing the Environment Abstraction Layer (EAL) and resetting the device controller.

Spool diagnoses the device failure type online through self-monitoring, analysis, and reporting technology (S.M.A.R.T.) data [35]. For media errors, the failure recovery component isolates the failed device so that the administrator can replace the failed device through hot-plug. All the other NVMe devices are unaffected by the failed device. For data link errors, the recovery component resets the device's controller directly, thus minimizing data loss due to unnecessary disk replacement.

We implement Spool based on the SPDK userspace NVMe driver. Spool combines the advantages of Virtio and VFIO (Figure 3). Furthermore, instead of implementing the actual Virtio datapath, we offload the datapath from QEMU to Spool adopting the vhost-user protocol. Adopting this protocol, the guest OS directly interacts with Spool without QEMU's intervention. In addition, by adopting the SPDK userspace polled driver specification [38], Spool eliminates the overhead of system calls and data copies between kernel space and userspace stacks on the host and achieves high I/O performance.

4 Reliable Cross-Process Journal

In this section, we describe the reliability problem in the Virtio protocol that virtualizes the NVMe device, and we present the design of a cross-process journal that improves reliability.

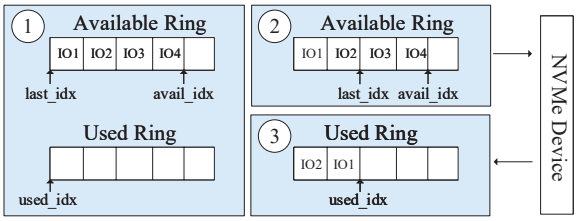


Figure 7: Design of Virtio block virtualization protocol.

4.1 Problem Statements

Figure 7 shows the design of the Virtio block driver that handles I/O requests in the guest OS. The I/O requests are processed in a *producer-consumer* model, where the guests are producers and the storage virtualization system is the consumer.

Specifically, the Virtio driver of each guest OS maintains an available ring and a used ring to manage its I/O requests. ① When an I/O request is submitted, the descriptor chain of the request is placed into the descriptor table. The descriptor chain includes the metadata, buffer, and status of the request. The metadata indicates the request type, request priority, and the offset of read or write. The guest driver places the index of the head of the descriptor chain into the next ring entry of the available ring, and the available index of the available ring ("`avail_idx`" in Figure 7) is increased. Then, the driver notifies the storage virtualization system that there is a pending I/O request. ② Meanwhile, the storage virtualization system running in the host obtains the several head indexes of the pending I/O requests in the available ring, increases the last index of the available ring ("`last_idx`" in Figure 7), and submits the I/O requests to NVMe device hardware driver. ③ Once a request is completed, the storage virtualization system places the head index of the completed request in the used ring and notifies the guest. Here, it is worth noting that the available ring and used ring are allocated by the guest, and the `avail_idx` is maintained by the guest, while both the `last_idx` and `used_idx` are maintained by the storage virtualization system.

The storage virtualization system may adopt either interrupt or polling to obtain I/O requests from the guest OS. Polling is able to fully utilize the advantages of NVMe devices to reap significant performance benefits [27, 37, 38], and Spool uses a dedicated I/O thread to poll I/O requests from the guest and data from the NVMe device instead of interrupts. This mechanism is implemented based on the SPDK userspace NVMe driver.

In general, the storage virtualization system runs well with the above procedure. However, if the storage virtualization system restarts for an upgrade or the NVMe device controller is reset, data loss may occur.

In the case of Figure 7, the storage virtualization system obtains two I/O requests, *IO1* and *IO2*. Then, the `last_idx` is incremented from *IO1* to *IO3* in the available ring. If the

storage virtualization system restarts at this moment, the last available index will be lost, which means that it does not know where to proceed with I/O requests after the restart. Even if the last available index persists, there is no way to know whether the obtained *IO1* and *IO2* have been completed. If we simply continue to process the next request based on the last available index, the previously obtained incomplete request will be lost.

When we reset the controller of an NVMe device, all the admin and I/O queue pairs are cleared. Suppose that *IO1* and *IO2* have been submitted but they are still in the device I/O queue and have not been processed. Due to the lack of an I/O request state, the submitted I/O request in the cleared I/O queue pairs will never be checked as completion from the NVMe device.

In summary, a journal is needed to maintain I/O consistency.

4.2 Design of the Journal

We propose a cross-process journal of data that persists in shared memory to solve the problem of data loss caused by the storage virtualization system restart or device controller reset. Spool persists the following data in the journal.

- Last available index (`last_idx`) of the available ring. The index references to the starting descriptor index of the latest request that the Virtio backend reads from the available ring.
- The head index of each request in the available ring. The index refers to the head of a descriptor chain in the descriptor table.

With the `last_idx` in the journal, Spool knows which requests have been processed after restarting for the upgrade and is able to continue processing the remaining to-be-processed requests. If a request's starting descriptor index is referenced between `last_idx` and `avail_idx` of the available ring, it is a to-be-processed request.

Algorithm 1 Algorithm of cross-process journal

```
Require: head1: The head index of request in the available ring;
Require: head2: The head index of completed request from the driver;
Require: req[]: A ring queue to mark each I/O reqeust in the journal;
Require: aux: A temporary union variable to record multiple variables;
 1: poll head1 from the available ring;
 2: aux.state = START;
 3: aux.last_idx = journal->last_idx+1;
 4: aux.last_req_head = head1;
 5: *(volatile uint64_t *)&journal->val = *(volatile uint64_t *)&aux.val;
 6: req[head1] = INFLIGHT;
 7: journal->state = FINISHED;
 8: submit I/O request to driver;
 9: poll head2 completion;
10: journal->usd_idx++;
11: req[head2] = DONE;
12: put head2 to the used ring, may goto 10 or 13;
13: update used_index of the used vring with usd_idx of journal;
14: req[head2] = NONE;
```

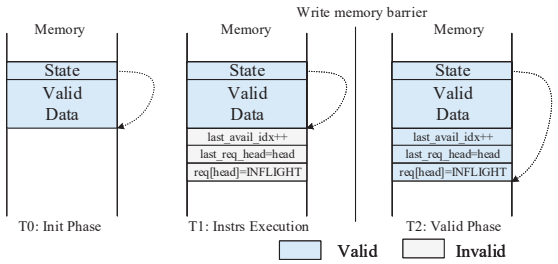


Figure 8: Transactional execution of multiple memory access instructions.

At the same time, when processing an I/O request in Spool, the request is given one of three states: INFLIGHT, DONE, or NONE. The cross-process journal uses Algorithm 1 to manage the I/O requests. To be more specific, when Spool gets a request from the frontend, it persists the head index of this request and marks the request as INFLIGHT, updates `last_idx`, and submits the request to the hardware driver. Once the I/O request completes, Spool updates the persisted `used_idx` in the journal and marks the request as DONE. After that, Spool returns the result of this request to the frontend, updates the used index of the frontend, and marks the request as NONE.

Adopting this method, if Spool restarts, the new Spool process can find out which request was not completed before the restart. In this way, the new Spool process resubmits the requests in the INFLIGHT state.

4.3 Merging Journal Update Instructions

An intuitive idea is to use shared memory as a journal to save this information with low latency overhead. However, it is challenging to ensure the consistency of the journal itself because Spool must update the journal multiple times during the processing of an I/O request.

Specifically, the process of each I/O request in Spool involves updating the last available index and marking the state of the request as INFLIGHT. During the processing, if Spool restarts or the controller is reset between the first two instructions, this request is lost.

If we can guarantee that instruction 3 (increase `last_idx`) and instruction 6 (change the request's status) in Algorithm 1 are executed in an atomic manner, the request loss problem can be resolved. However, only reading or writing a quadword aligned on a 64-bit boundary is guaranteed to be carried out atomically [7] in the memory, and the two instructions are not atomic when operating on the cross-process journal that resides in the memory.

To resolve the above problem, we design a multiple-instruction transaction model to guarantee atomic execution of the two instructions. As shown in Figure 8, each transaction consists of three phases. In T0, the init phase, we make a copy of the variable to be modified, such as `last_idx`, and in T1,

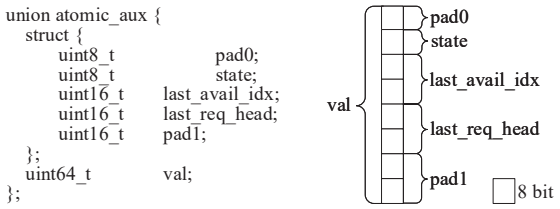


Figure 9: Aux data structure that enables the update of multiple indexes using a single instruction.

the transaction will be in the START state. After all the instructions complete, the transaction will be in the FINISHED state in T2. Because the state is guaranteed to be carried out atomically, once the last available index updates, the related request is recorded as INFLIGHT in the journal. If any failure occurs in one transaction, we rollback the transaction to erase all data modifications with the copy.

As shown in Figure 9, we also design an auxiliary data structure carefully to eliminate the overhead of making a copy in T0 using a union type. The state, last available index, and head index of the related request are padding to 64 bits and a union memory block with a 64-bit value. We could update these three records within one instruction in Algorithm 1 step 5. This is a valuable trick to efficiently maintain journal consistency.

4.4 Recovering I/O Requests from Journal

Spool uses Algorithm 2 to recover the unprocessed I/O requests before the restart. With multiple journal transactions and an auxiliary structure, the new Spool process before the restart only needs to check the state and decide whether to redo the transactions or not.

Algorithm 2 Algorithm for recovering I/O requests

```
1: if (state == START) {
2:       req[last_get_req_head] = INFLIGHT;
3:       state = FINISHED;
4: }
5: jstate = (last_used_idx == used_idx) ? NONE : INFLIGHT
6: change all requests with done status to jstate;
7:
8: last_used_idx = used_idx;
9: submit all requests marked as INFLIGHT;
```

The recovery algorithm works based on the value of the used index of vring and the last used index in the journal. If they are equal, Spool may crash after step 13 in Algorithm 1, but we do not know whether step 14 completes. Therefore, Spool tries to execute step 14 again and changes the states of the DONE requests to NONE. Otherwise, the request's process may be broken between steps 10 and 12. In this case, we do not know whether the request in the state DONE has been submitted to the frontend. To avoid losing any I/O request, we roll back the status of all the DONE requests to INFLIGHT. Because the frontend always has correct data,

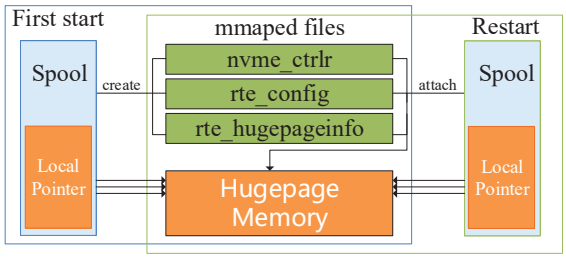


Figure 10: Boosting the restart of Spool by reusing the stable configurations.

we synchronize the last used index in the journal with the frontend used index. In the last step, Spool resubmits all the requests that are in the INFLIGHT state.

Now, in Spool, the size of a single journal is only 368 bytes because it only records the metadata and the indexes of the requests in the available ring. Note that the above algorithm does not take precautions against the journal wrapping around; this is not possible because the journal is the same size as the available ring, so the guest driver will prevent such a condition.

5 Optimizing Spool Restart

As shown in Figure 5, when restarting a storage virtualization system, it initializes the EAL (Environment Abstraction Layer) in the DPDK driver and probes the NVMe devices in the SPDK driver on the host node. The relatively long restart time (ranging from 450 ms to 2,500 ms) hurts the availability as the whole storage system is out of service before the restart completes. In this section, we describe the method of optimizing the restart procedure in Spool.

5.1 Reusing Stable Configurations

During EAL initialization, the DPDK driver reserves all the memory in an IOVA-contiguous manner and sets up the huge pages for memory usage, such as I/O request buffers that are shared by the host userspace datapath and the physical device to perform DMA transactions. To be more specific, DPDK maps all the available huge pages to the process address space of DPDK, reads "`/proc/self/pagemap`" from the host OS to find the corresponding physical addresses of the huge pages, and sorts and merges the physical addresses into large contiguous chunks. After that, DPDK uses the physically continuous huge page chunks as memory segments. This design choice enables better hardware data prefetching and results in higher communication speed. Our experiment shows that the whole time spent on obtaining the physical memory layout information of huge pages is approximately 800 milliseconds, accounting for 70.9% of the total down time for a Samsung PM963 NVMe device. In our cloud, all the SSDs are restarted by more than 800,000 times in total

in a single year. More restarts are required to periodically update the SPDK driver or apply new security patches. The long EAL initialization results in a long restart time and poor tenant experience.

Based on the above findings, we optimize the initialization steps of Spool. Specifically, the new Spool process after restart reuses the memory layout information from the current Spool process. Figure 10 shows the way we enable memory layout reuse. As shown in the figure, after the first startup of Spool, we store the related information (e.g., the huge pages in use and the virtual addresses of the huge pages) in the memory-mapped files that reside in the memory. If Spool restarts, it directly obtains the required information from the memory-mapped files in the memory with short latency and operates normally. Specifically, the "`rte_config`" file stores the global runtime configurations, and the "`ret_hugepageinfo`" file stores the memory layout information related to the huge page chunks used by Spool.

The above design does not guarantee that the new Spool after the restart will still use the largest continuous physical memory. This is because other processes may release huge pages and form larger continuous physical memory chunks. This design choice eliminates the long scan time of huge pages and does not degrade the performance of Spool.

5.2 Skipping Controller Reset

When probing the NVMe devices during the restart of the SPDK driver, more than 90% of the time is spent resetting the controller of NVMe devices. On an Intel P3600 SSD, the NVMe probe stage takes more than 1500 milliseconds (Figure 5). During the reset of the controller, SPDK frees the current admin queue, the I/O queue[1] in the controller, and creates them again for the controller after the reset.

Compared with SPDK, Spool skips the controller reset step during restart and reuses the data structures of the controller. This design is valid because the restart of Spool is not caused by media errors or data link errors. In this case, the data structures of the NVMe device controllers are not broken. To achieve reuse, Spool saves the NVMe device controller-related information in the memory-mapped file "`nvme_ctrlr`", as shown in Figure 10). After Spool restarts, it reuses the data of the device controller.

The challenging part here is that the context of the I/O request has disappeared with the exit of Spool. Therefore, we need to ensure the admin queue and I/O queue are completely clean.

In general, various signals are used to terminate one running process for an OS, such as *SIGTERM*, *SIGINT*, and *SIGKILL*. The default action for all of these signals is to cause the process to terminate. To gracefully terminate, we catch the termination signals including *SIGTERM* and *SIGINT* to ensure the completion of all INFLIGHT I/O requests before actually terminating. In that case, we could skip the reset operation after restart. Regarding the *SIGKILL* signal, which could not be handled in process or abnormal exit, we reset the controller after restart as usual.

[1]There are two types of commands in NVMe: Admin Commands are sent to Admin Submission and Completion Queue. I/O Commands are sent to I/O Submission and Completion Queues [9].

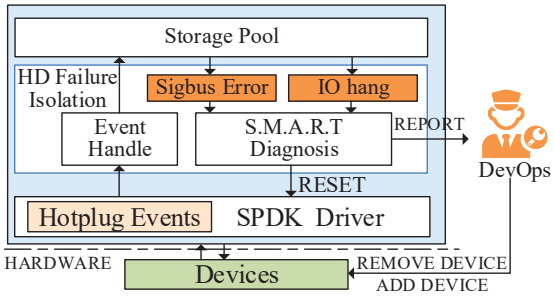


Figure 11: Handling hardware failures in different ways.

6 Hardware Fault Diagnosis and Processing

Traditionally, any NVMe device hardware failure causes the whole machine to be offline and repaired, and all VMs on the failing node need to be proactively migrated to a healthy node. With emerging virtualized storage systems like SPDK, to fix an NVMe device failure on a node, the administrator directly replaces the failed device through hot-plug. On one hand, this causes data loss for users, and on the other hand, it increases operating costs.

To minimize data loss and reduce operating costs, we have implemented a fault diagnosis to identify the type of hardware fault and effectively avoid unnecessary hardware replacement.

6.1 Handling Hardware Failures

In large-scale cloud-based productions, hardware failures are frequent and may cause *SIGBUS* error and I/O hang, as shown in Figure 11.

Spool adopts the SPDK userspace NVMe driver to access a local NVMe PCIe SSD. Base address register (BAR) space for the NVMe SSD will be mapped into the user process through VFIO, which allows the driver to perform MMIO directly. The BAR space will be accessed by Spool while guest VMs send I/Os to the devices. However, the BAR space may become invalid while the device fails or is hot-removed. At this time, it will trigger *SIGBUS* error and cause the host process to crash if guest VMs still send I/O requests to the failed device of the host.

To improve reliability, a *SIGBUS* handler is registered into Spool. Once guest VMs send I/O requests to failed devices and access illegal BAR space, the handler will capture the *SIGBUS* error and remap the invalid BAR space to a dummy virtual address so that the *SIGBUS* error will not be triggered

Table 1: Experimental configuration

Host configuration	
CPU & Memory	2x E5-2682v4 @2.5GHz; 128GB DDR4 Memory
NVMe devices	2 Samsung PM963 3.84TB SSDs
OS info	CentOS 7 (kernel verison 3.10.327)
Guest OS configuration	
CPU & Memory	4 vCPU; 8 GB
OS info	CentOS 7 (kernel verison 3.10.327)

again. Then, it sets the NVMe controller state to failure and fails all the internal requests in NVMe qpairs of the failed device.

6.2 Failure Model

The S.M.A.R.T. diagnosis collects and analyzes S.M.A.R.T. data to identify the failure type. S.M.A.R.T. data is disk-level sensor data provided by the firmware of the disk driver, including smart-log, expanded smart-log, and error-log, which can be used to analyze internal SSD errors.

Once a hardware media error is verified, Spool proactively fails the submitted I/O requests and returns I/O errors. After that, all the subsequent I/O requests to the failed device will return errors to guest VMs directly. Meanwhile, the S.M.A.R.T. diagnosis will send a report to DevOps. The hot-plug feature in the driver layer of SPDK is utilized in Spool; hence, the failed device can be replaced directly.

For the other hardware errors, such as a data link layer failure, diagnosis informs Spool to reset the controller. During the reset process, the I/O requests from the guest VMs hang. After the device is fixed, the INFLIGHT requests in the journal are resubmitted automatically.

7 Evaluation of Spool

In this section, we evaluate the performance of Spool in resolving hardware failure and supporting live upgrades. We first describe the experimental setup. Then, we evaluate the reliability of Spool in fixing an NVMe device's hardware failure without affecting other devices and correctly tolerating system upgrades at random times. After that, we show the effectiveness of Spool in reducing the system restart time, followed by a discussion on the impact of Spool on the I/O performance and the extra overhead caused by the cross-process journal. Lastly, we discuss the effectiveness of Spool on an in-production large-scale public cloud.

7.1 Experimental Setup

We evaluated Spool on a server equipped with two Intel Xeon E5-2682 processors operating at 2.5 GHz with 128 GB memory. For the NVMe devices, we adopted a mainstream SSD device: Samsung PM963 NVMe SSD. Table 1 summarizes the hardware and software specifications of the experimental platform.

For extensive evaluation, we use the Flexible I/O tester (FIO) [6] as our performance and reliability evaluation benchmark. FIO is a typical I/O tool meant to be used both for benchmark and stress/hardware verification and is widely used in research and industry. When evaluating the I/O performance, we use different parameters, as shown in Table 2, to demonstrate metrics, including IOPS and average latency recommended by Intel [5] and Alibaba [4]. To emulate the real-system cloud scenario, we split each NVMe SSD into three partitions (each partition was 100 GB) and allocated each partition to an individual VM.

We used libaio (Linux-native asynchronous I/O facility) [21] as the FIO load generation engine. Table 2 lists the generated FIO test cases. To obtain accurate performance, we tested raw SSDs without any file system.

Table 2: FIO test cases

Tested metrics	Test cases	FIO Configuration (bs, rw, iodepth, numjobs)
Bandwidth	Read	(128K, read, 128, 1)
	Write	(128K, write, 128, 1)
IOPS	Randread	(4K, randread, 32, 4)
	Mixread	(4K, randread 70%, 32, 4)
	Mixwrite	(4K, randwrite 30%, 32, 4)
	Randwrite	(4K, randwrite, 32, 4)
Average Latency	Randread	(4K, randread, 1, 1)
	Randwrite	(4K, randwrite, 1, 1)
	Read	(4K, read, 1, 1)
	Write	(4K, write, 1, 1)

7.2 Reliability of Handling Hardware Failure

When an NVMe device suffers from hardware failure, Spool isolates the failed device and performs device replacement or controller reset accordingly. When handling such failure, Spool should not affect the I/O operations on other devices of the same node, and the VMs that are using the failed device should receive I/O errors instead of exiting abnormally.

We designed an experiment to evaluate Spool in the above scenario. In the experiment, we launched two VMs on a hardware node equipped with two NVMe devices and configured the two VMs to uses different NVMe devices in Spool. The two VMs randomly read data from NVMe devices in the beginning, and we manually removed an NVMe device and observed the behavior of the two VMs.

Figure 12 presents the I/O performance of the two VMs when the NVMe device ("SSD2") was hot-removed at time 80 s. The hot remove was performed by writing a non-zero value to "`/sys/bus/pci/devices/.../remove`". Observed from this figure, the I/O performance of VM1 that uses the NVMe device SSD1 is not affected when SSD2 is removed. Meanwhile, VM2 does not exit abnormally after SSD2 is removed. Once a new SSD device replaces the failed SSD2 or the controller of SSD2 is reset correctly at time 95 s, VM2 is able to directly use SSD2 without any user interference.

Spool can successfully handle the above hardware failure because it catches the hardware hot-plug event and diagnoses the device failure type first. Hardware failures are handled

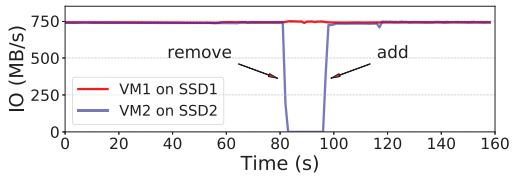


Figure 12: Handling hardware failure with Spool.

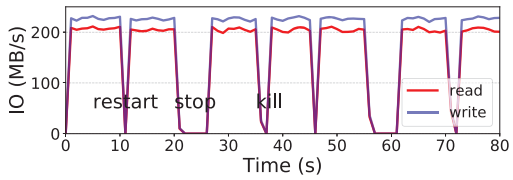


Figure 13: Data consistency at live upgrade with Spool.

in two ways: media errors are solved by hot-plugging a new SSD, and then the storage service automatically recovers, where the related logical devices are automatically mapped to new devices, while data link failure is handled by controller reset instead of replacing a SSD. On the contrary, if the traditional SPDK is used to manage SSDs, the hardware failure can only be solved by hot-plugging new devices, resulting in unnecessary data loss, and the storage service of SPDK needs to be reset manually for recovery.

7.3 Reliability of Handling Random Upgrades

To validate the reliability of Spool in handling upgrades without resulting in data loss, we designed an experiment that *restarts* purposely *stops and starts* and randomly *kills and restarts* Spool. In the experiment, we relied on the data verification function in FIO to check the data consistency. By enabling the data verification function, FIO verifies the file contents after writing 10 blocks contiguously with crc32 and reports whether any data corruption occurred or not. If FIO runs completely without errors, data consistency is verified.

Figure 13 shows the read and write performance to the SSD when we restart on purpose, stop and start, and randomly kill and restart Spool at time 10 s, 20 s, and 35 s. As can be observed from the figure, the I/O operations to the SSD complete correctly with Spool, even if Spool is directly killed and restarted for an upgrade.

Spool can guarantee data consistency during upgrades due to the cross-process journal. The journal persists the current states of all the NVMe devices. Whenever Spool is restarted, it is able to recover the states before the restart and continue to complete the unprocessed I/O requests. On the contrary, with SPDK, there is no mechanism to guarantee data consistency for INFLIGHT I/Os.

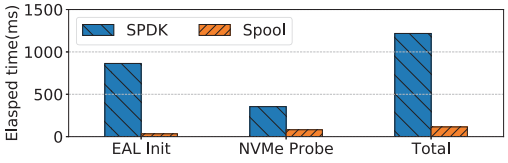


Figure 14: Restart times of Spool and SPDK.

7.4 Reducing Restart Time

Observed from Figure 13, the downtime due to the restart is short with Spool. In more detail, Figure 14 shows the restart time breakdown of Spool and SPDK.

As can be observed from this figure, Spool significantly reduces the total restart time from 1,218 ms to 115 ms on a Samsung PM963 SSD. This significant restart time reduction originates from the reduction of EAL initialization time and the NVMe probe time.

SPDK suffers from a long EAL initialization time and a long NVMe probe time because it initializes the EAL and resets the controller of the device during probing at each startup. By reusing the previous memory layout information, Spool minimizes the EAL initialization time. And, by skipping resetting the device controller, Spool reduces the NVMe probe time.

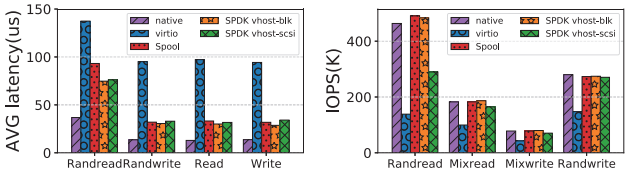
7.5 I/O Performance of Spool

It is crucial to ensure high I/O performance (i.e., high IOPS and low latency) when guaranteeing reliability. In this subsection, we report the I/O performance of NVMe devices with Spool in two cases: an NVMe device is only allocated to a single VM, and an NVMe device is shared by multiple VMs.

7.5.1 Case 1: Single VM Performance

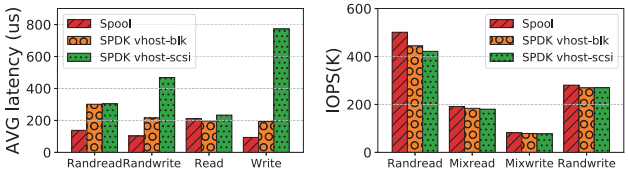
Figure 15 presents the data access latency and IOPS to an SSD when it is virtualized with Virtio, SPDK, and Spool. In the figure, "native" shows the performance of the SSD measured on the host node directly; "SPDK vhost-blk" and "SPDK vhost-scsi" show the performance of the SSD if SPDK is used as the I/O virtualization system and the SSD is treated as a block device or a SCSI device, respectively.

As can be observed from Figure 15, all the I/O virtualization systems result in longer data access latency compared with the native access due to extra layers in the virtualization system. Meanwhile, Spool achieves similar data access latency to SPDK. From the IOPS aspect, the IOPS of *Randread* with Spool is 2.54x higher than Virtio and even slightly better than the native bare metal. As mentioned in Section4, Spool uses polling instead of interrupt to monitor the I/O requests. Polling saves the expense of invoking the kernel interrupt handler and eliminates context switching. These results are consistent with prior work [27].



(a) Average latency (b) IOPS

Figure 15: Average data access latency and IOPS of an NVMe SSD when it is used by a single VM.



(a) Average latency (b) IOPS

Figure 16: Average data access latency and IOPS of an NVMe SSD when it is shared by multiple VMs.

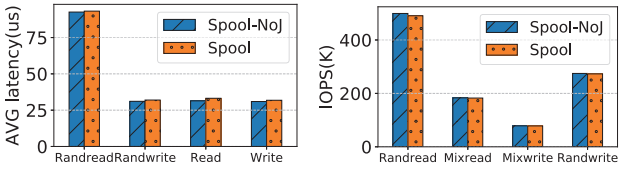
Compared with SPDK vhost-blk, the performance of our implementation is almost the same. Because the SPDK vhost-blk software stack is thinner than SPDK vhost-scsi, the IOPS with SPDK vhost-scsi is lower than that with Spool.

7.5.2 Case 2: Scaling to Multiple VMs

In this experiment, we partition an SSD into three logic disks and assign each logic disk to an individual VM. This experiment tests the effectiveness of Spool in handling multiple VMs on an NVMe device.

Figure 16 shows the IOPS and data access latency when three VMs share an NVMe device, and each result is the sum of those of all VMs. For the latency test, we ran each benchmark 10 times and report the average latency for each benchmark. As can be observed from this figure, Spool does not degrade the I/O performance of all the benchmarks compared with SPDK vhost-blk and SPDK vhost-scsi. Specifically, Spool improves the IOPS of *Randread* by 13% compared with SPDK vhost-blk, which reduces the average data access latency of *Randread* by 54% to 55% compared with SPDK vhost-blk and SPDK vhost-scsi, respectively.

Besides, we can see that the SSD device achieves similar IOPS when the SSD is used by a single VM and three VMs, and we can see the average data access by comparing Figure 15(b) and Figure 16(b). The data access latency of the benchmarks when the SSD is shared by three VMs is three times that of case 1. This is reasonable because the backend I/O load pressure increases linearly with the number of VMs, so the total latency of the three VMs increases. While the I/O load pressure of one VM has reached the throughput limit of the Samsung PM963 specification [10], the total IOPS of three VMs remains unchanged. The I/O performance of Spool is slightly better than that of SPDK because Spool and



(a) Average latency (b) IOPS

Figure 17: Overhead of the cross-process journal.

SPDK use different logical volume management mechanisms. Specifically, Spool uses the buddy system to manage logical volumes, while SPDK uses Blobstore.

7.6 Overhead of the Cross-Process Journal

To measure the overhead of the cross-process journal, we implement a Spool variation, Spool-NoJ that disables the cross-process journal. Figure 17 shows the data access latency and the IOPS of the SSD with Spool and Spool-NoJ.

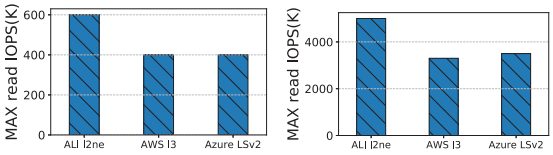
As shown in Figure 17, Spool-NoJ and Spool result in similar data access latency and IOPS. Compared with Spool-NoJ, Spool increases the average data access latency no more than 3%. Meanwhile, Spool reduces the IOPS by less than 0.76% compared with Spool-NoJ. The extra overhead caused by the cross-process journal in terms of average latency and IOPS throughput is negligible.

7.7 Deployment on an In-production Cloud

We currently deploy Spool in 210 clusters with approximately 20,000 physical machines equipped with approximately 200,000 NVMe SSDs.

On the cloud supported by Spool, we built a Platform-as-a-Service cloud(ALI I2 series) that provides low latency, high random IOPS, and high throughput I/O support. The maximum IOPS of single disk is 50% higher than that of competitive products and the maximum IOPS of a largest specification instance is 51% higher than that of the competitive products, as shown in Figure 18. The in-production cloud hosts Cassandra, MongoDB, Cloudera, and Redis. They are ideal for Big Data, SQL, NoSQL databases, data warehousing, and large transactional databases. Recently, the instance resources of local SSD disks have helped OceanBase break the world record for TPC-C benchmark performance test maintained by Oracle for 9 years and becoming an important milestone in the evolution history of global databases.

A holistic fine-grain monitoring system is crucial for clouds. While the monitoring system is able to diagnose media errors and other SSD failures, Spool handles the failures in different ways. Our statistics show that the current hardware failure rate is approximately 1.2% over a whole year. Throughout one year, approximately 2,400 out of 200,000 SSDs suffer from media errors. The media error is due to either media damage or life depletion. From the system upgrade aspect,



(a) A single disk (b) A single instance

Figure 18: Maximum read IOPS compared with AWS and Azure.

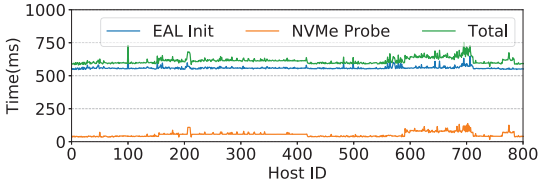


Figure 19: Restart time of Spool during a live upgrade.

we release a new version of Spool every six months. In total, we upgrade Spool on more than 40,000 physical machines every year. The purpose of the new version is to deal with two issues: 1) releasing new features and 2) fixing online stability during the operation and maintenance phase. Most of the new features are related to performance, such as adding support for multiple queues, optimizing memory DMA operation, and optimizing memory pools.

Figure 19 shows the restart time of some selected machines in a live upgrade in production. The *x*-axis is the ID of the physical node. Due to historical reasons, Spool in the production environment is based on the earlier version of the DPDK driver, and the initialization memory requires 1,172 ms, accounting for 70% of the initialization of EAL (1,792 ms in total), which is almost optimized. However, the rest of the initialization of the EAL is still 550 ms, and the total upgrade time for 95% of the machines is within 654 ms. We are working on updating the DPDK driver for Spool.

8 Related Work

There has been a lot of work concentrating on NVMe virtualization and optimizing storage I/O stacks for modern fast storage devices(e.g., NVMe SSDs).

Kim et al. [22] analyzed the overheads of random I/O over storage devices in detail and mainly focused on optimizing the I/O path by eliminating the overhead of user-level threads, bypassing the 4KB aligned I/O routine and enhancing the interrupt delivery delay in QEMU. In QEMU/KVM forum 2017, Zheng et al. [43] implemented a userspace NVMe driver in QEMU through VFIO to accelerate the virtio-blk in a guest OS at the cost of device sharing. Peng et al. [27] discussed the importance of polling for NVMe virtualization and took advantage of polling to achieve extreme I/O performance while each polling thread brings 100% usage of 3 cores.

Virtio [12] is a de facto standard for para-virtualized driver specifications including virtio-blk and virtio-scsi, which defines the common mechanisms for virtual device discovery and layouts. However, each I/O request passes through the I/O stack twice for guest and host, causing great loss of I/O performance. Then, a vhost acceleration method is proposed to accelerate virtio-scsi or virtio-blk provided by the storage performance development kit (SPDK) [38], such as kernel vhost-scsi, userspace vhost-scsi, and vhost-blk.

While local SSDs provide higher data access bandwidth and lower latency, storage disaggregation (e.g., ReFlex [24], NVMe-over-Fabrics [20], [23]) enables flexible scaling and high utilization of Flash capacity and IOPS at the cost of interconnecting latencies over a network. The bandwidth of NIC is the bottleneck to saturate the bandwidth of multiple disks. Two NVMe SSDs may saturate the bandwidth of emerging NIC (100GbE). Meanwhile, the upgrade of cloud services with minimal downtime has also been widely studied. Neamtiu et al. [26] highlighted challenges and opportunities for upgrades to the cloud. Clark et al. [19] proposed a live migration mechanism to temporarily move VMs to a backup server, upgrade the system, and then moves the VMs back. Zhang et al. [41] proposed Orthus to live upgrade the VMM without interrupting customer VMs and significantly reduce the total migration time and downtime. However, Orthus only focuses on KVM and QEMU and ignores the backend services.

9 Conclusion

This paper presented Spool, a holistic virtualized storage system that is capable of handling hardware failure and the NVMe virtualization system upgrades reliably. Spool significantly reduces the restart time by 91% on a Samsung PM963 SSD by reusing the data structures at the restart for system upgrades. Compared with emerging virtualized storage systems such as SPDK, Spool supports live upgrades and guarantees data consistency with a shared memory-based journal at any time. Moreover, Spool diagnoses device failure type instead of hot-plugging directly and eliminates most unnecessary NVMe device replacement.

Acknowledgement

This work is partially sponsored by the National R&D Program of China (No. 2018YFB1004800), the National Natural Science Foundation of China (NSFC) (61632017, 61772480, 61872240, 61832006, 61702328). Quan Chen and Minyi Guo are the corresponding authors. We thank Lingjun Zhu for his help in the experiments. And we also thank Mike Mesnier for shepherding our manuscript.

References

[1] Alibaba cloud instance family with local ssds. https://www.alibabacloud.com/help/doc-detail/25378.htm?spm=a2c63.p38356.879954.7.158f775aPotRZi#i2.

[2] Amazon ec2 i3 instances. https://aws.amazon.com/ec2/instance-types/i3/.

[3] Azure lsv2-series. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/sizes-storage#lsv2-series.

[4] Block storage performance. https://www.alibabacloud.com/help/doc-detail/25382.htm?spm=a2c63.p38356.879954.28.344791f3OdgBQZ#title-1rp-8na-22y.

[5] Evaluate performance for storage performance development kit. https://software.intel.com/en-us/articles/evaluate-performance-for-storage-performance-development-kit-spdk-based-nvme-ssd.

[6] Fio. https://fio.readthedocs.io/en/latest/.

[7] Intel® 64 and ia-32 architectures software developer's manual. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.pdf.

[8] Local ssd. https://www.alibabacloud.com/help/doc-detail/63138.htm?spm=a2c63.p38356.879954.199.aa265e6dMUz7fg#concept-g3w-qzv-tdb.

[9] Nvm express specification. http://www.nvmexpress.org/specifications.

[10] Pm963 specifications. https://www.samsung.com/semiconductor/ssd/enterprise-ssd/MZQLW3T8HMLP/.

[11] Utilizing local nvme storage on azure. https://docs.microsoft.com/en-us/azure/virtual-machines/linux/storage-performance#utilizing-local-nvme-storage.

[12] Virtio homepage. https://www.linux-kvm.org/page/Virtio.

[13] Vmware homepage. https://www.vmware.com.

[14] Xen main page. https://wiki.xen.org/wiki/Main_Page.

[15] Amro Awad, Brett Kettering, and Yan Solihin. Non-volatile memory host controller interface performance analysis in high-performance i/o systems. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 145–154. IEEE, 2015.

[16] Sushil Bhardwaj, Leena Jain, and Sandeep Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[17] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. Live updating operating systems using virtualization. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 35–44. ACM, 2006.

[18] Quan Chen and Qian-ni Deng. Cloud computing and its key techniques [j]. *Journal of Computer Applications*, 9(29):2562–2567, 2009.

[19] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[20] Zvika Guz, Harry Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–9, 2017.

[21] William K Josephson. An introduction to libaio. 2007.

[22] Jungkil Kim, Sungyong Ahn, Kwanghyun La, and Wooseok Chang. Improving i/o performance of nvme ssd on virtual machines. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 1852–1857. ACM, 2016.

[23] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. Flash storage disaggregation. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 29. ACM, 2016.

[24] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. *ACM SIGARCH Computer Architecture News*, 45(1):345–359, 2017.

[25] Uri Lublin, Yaniv Kamay, Dor Laor, and Anthony Liguori. Kvm: the linux virtual machine monitor. 2007.

[26] Iulian Neamtiu and Tudor Dumitraş. Cloud software upgrades: Challenges and opportunities. In *2011 International Workshop on the Maintenance and Evolution*

of Service-Oriented and Cloud-Based Systems, pages 1–10. IEEE, 2011.

[27] Bo Peng, Haozhong Zhang, Jianguo Yao, Yaozu Dong, Yu Xu, and Haibing Guan. Mdev-nvme: a nvme storage virtualization solution with mediated pass-through. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 665–676, 2018.

[28] James L Peterson and Theodore A Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

[29] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.

[30] Jeffrey Shafer. I/o virtualization bottlenecks in cloud computing today. In *Proceedings of the 2nd conference on I/O virtualization*, pages 5–5. USENIX Association, 2010.

[31] Sankaran Sivathanu, Ling Liu, Mei Yiduo, and Xing Pu. Storage management in virtualized cloud environment. In *2010 IEEE 3rd International Conference on Cloud Computing*, pages 204–211. IEEE, 2010.

[32] Yongseok Son, Hyuck Han, and Heon Young Yeom. Optimizing file systems for fast storage devices. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 8. ACM, 2015.

[33] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson, et al. {DC} express: Shortest latency protocol for reading phase change memory over {PCI} express. In *Proceedings of the 12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 309–315, 2014.

[34] Wikipedia contributors. Serial ata — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Serial_ATA&oldid=932414849, 2019. [Online; accessed 26-December-2019].

[35] Wikipedia contributors. S.m.a.r.t. — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=S.M.A.R.T.&oldid=931348877, 2019. [Online; accessed 8-January-2020].

[36] Alex Williamson. Vfio: A user's perspective. In *KVM Forum*, 2012.

[37] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

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

[39] Ziye Yang, Changpeng Liu, Yanbo Zhou, Xiaodong Liu, and Gang 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)*, pages 67–76. IEEE, 2018.

[40] Xiantao Zhang and Yaozu Dong. Optimizing xen vmm based on intel® virtualization technology. In *2008 International Conference on Internet Computing in Science and Engineering*, pages 367–374. IEEE, 2008.

[41] Xiantao Zhang, Xiao Zheng, Zhi Wang, Qi Li, Junkang Fu, Yang Zhang, and Yibin Shen. Fast and scalable vmm live upgrade in large cloud infrastructure. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 93–105. ACM, 2019.

[42] Yiying Zhang and Steven Swanson. A study of application performance with non-volatile main memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10. IEEE, 2015.

[43] Fam Zheng. Userspace nvme driver in qemu. In *KVM Forum 2017*, pages 25–27, 2017.