Next generation storage architecture

Background and motivation

When it comes to storage, EVE's primary goal is to take raw, mutable storage devices (SSD, eMMC, NVMe drives, etc.) and provide:

1. file system abstraction for all system-level services running on EVE so that they can have some amount of mutable state (e.g. a newlog service requiring a mutable, persistent storage to keep gzip'ed log files for as long as it takes to transmit them back to the controller)
2. mutable layers supporting OCI containers
3. mutable disk images supporting VMs

The current implementation of mutable storage management in EVE is rather simplistic: it takes a single device, formats it with ext4 (with support for encryption turned on) and makes the resulting filesystem available under /persist. This satisfies #1 in the most basic POSIX filesystem sense and it makes both #2 and #3 be handled by qemu emulating either POSIX filesystem or block devices on top of raw qcow2/raw files under /persist. While this approach got us this far, the following areas clearly need improvement:

- ***multiple physical drive support***: an increasing number of servers are provisioned with multiple drives and all of them need to be stitched together to provide a seamless support for #1-#3 use cases listed above. Note that this is different from hot-provisioning (we are yet to see that kind of a use case) or hot-swapping of the drives – we're simply talking about making use of all the mutable storage available in the system.
- ***lack of quotas***: all 3 of use cases listed above suffer from a potential issue of a runway process consuming all of the available storage space starving the rest of EVE
- ***lack of ubiquitous, transparent compression and thin provisioning***: while we have bits and pieces of it (e.g. we support compressed qcow2 images) it is not enough to support all of the 3 use cases listed above
- ***performance degradation with multiple VMs running on the system***
- ***CPU and RAM consumption with multiple VMs running on the system***

While the last two items on the list ended up being an ultimate motivation (and key objectives) for this proposal, we should keep the entire list in mind when we discuss our next generation storage architecture.

Taking a purely performance- and resource-centric view, quickly leads to an observation that the biggest issue we are facing is multiple qemu user space processes trying to provide block caching layer for each VM that runs on EVE. To reiterate: the problem is not qemu per-se (it is still very much required to setup guest VMs), but rather its block caching and disk emulation layer that:

- proved to be notoriously difficult to optimize even for a single VM
- has no awareness of other qemu processes and their block caching layers
- doesn't play nice with cgroups

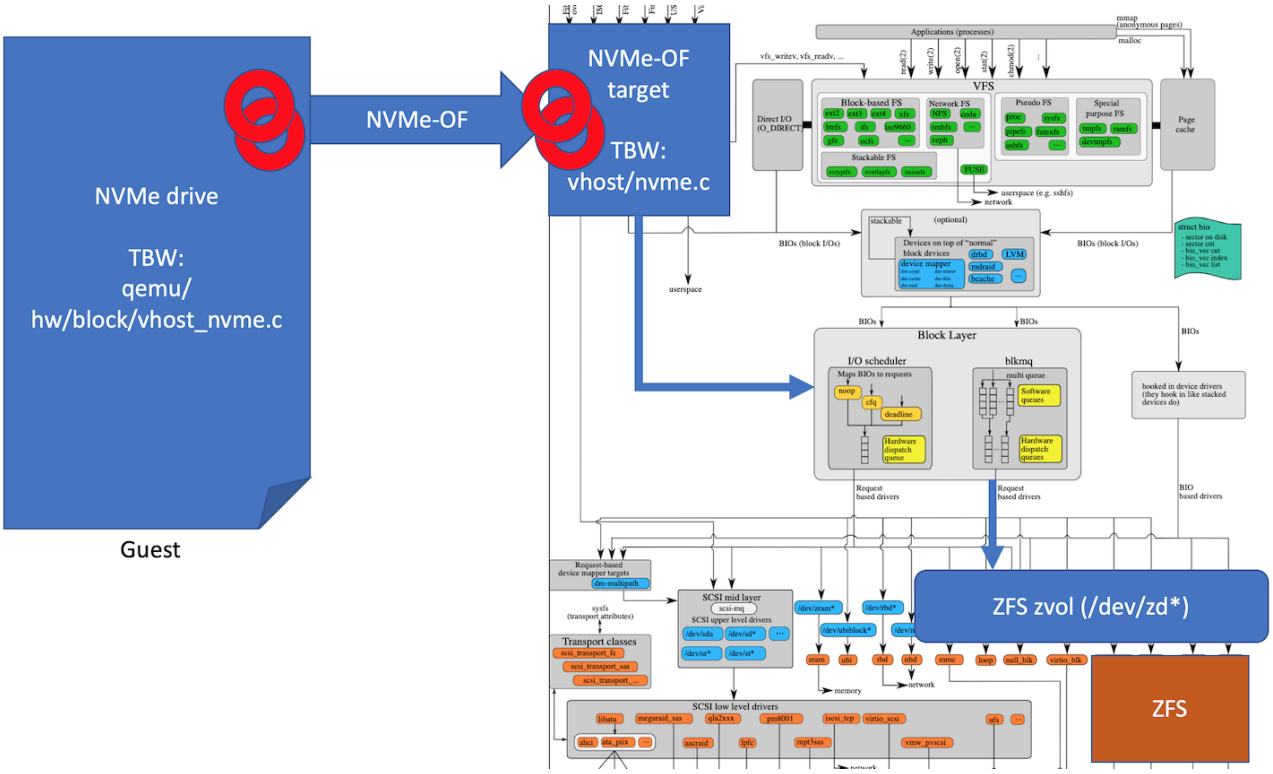
It seems then, that the most rational solution is to get rid of qemu block management layer altogether and make sure that the block I/O coming from guests terminates in a unified block management layer provided either by the Linux kernel itself or a single daemon process running on the system.

While we first considered a daemon process as our preferred option, it very quickly became apparent that the closest contender (SDPK) is still not mature and optimized enough to provide desired benefits. We may come back to it at some point, but for now we decided to focus on the Linux kernel itself.

After all, Linux kernel already has a very robust and well optimized BIO layer (with battle tested I/O schedulers) and a number of in-kernel targets that can turn guest I/O traffic into Linux kernel native BIO requests. After considering two of these targets: SCSI and NVMe, we've decided to focus on NVMe as it is a much cleaner protocol with a huge upside of being extremely well suited for parallelization (32 I/O queues for SCSI vs. 64k queues for NVMe).

Proposal

Our proposal is to standardize on NVMe-OF protocol as a Guest Host storage I/O communication protocol and use existing in-kernel NVMe-OF target to translate it into the native Linux kernel BIO requests. Once that is done, BIO requests can flow into either LVM or ZFS block devices with those providing all the features like thin provisioning, compression and encryption that are missing in the raw block device implementation. This will get us on-par with the current qemu implementation and its usage of qcow2 files for backing store, but may require either on-the-fly conversion of qcow2 images OR agreeing to use other image formats for initial content trees (e.g. ZFS snapshot streams). The overall flow of block I/O traffic can be summarized in the following picture:



While a lot of the building blocks required for this are already available in upstream Linux kernel and qemu the following are still missing:

- qemu NVMe driver that will route guest's NVMe traffic to virtio fabric (although previous attempts at upstreaming this exist)
- virtio fabric driver in the Linux kernel's NVMe target (although previous attempts at upstreaming this exist and there's a very similar SCSI driver already upstreamed)

Development steps

1. Introduce HV_EXPERIMENTAL (exactly what HV_HVM, but with NVMe-OF for disk) - Roman
2. Start with nullblk as a backend (potentially loopback) - Vitaliy
3. Resurrect vhost PRs in pillar, etc. - Vitaliy
4. Add checksums to fio test container - Vitaliy
5. Implement vhost/nvme.c in Linux kernel - Sergey
6. Implement vhost_nvme.c driver in qemu - Vitaliy
7. Build a test rig for actual NVMe drives - Dima

Discussion

The nice property of this design is that we can start with independent bits and pieces and keep an eye on performance. For example, without writing any additional code we can send NVMe traffic via TCP from our guests to the host helping us test assumptions about scalability (of course, this is expected to be much slower than virtio/vhost implementation, but is available in a stock kernel today).

It must also be noted, that while experimenting with vhost and SPDK a few EVE PRs that can be re-used were generated and will be repurposed for this work.

It was suggested by one of our community members that "would be great to avoid any bio repacking on host userspace side and make it possible to fetch nvmeof requests directly on the host kernel side (since guest and host share the same memory, that can be doable). Yes, that means you write your own driver for the host as well, but still you do that for the guest, so should not be a big problem"