Copy Offload

Here Be Dragons

Martin K. Petersen

martin.petersen@oracle.com

2019-08-21

SCSI Copy Offload

- Two variants:
  - `EXTENDED COPY` command sent to either source or destination describing which block ranges to copy from one to the other.
  - `POPULATE TOKEN` receives a token from the source device. The token is a cookie which represents the data to be sent somewhere else. The token is then sent to the destination device via `WRITE USING TOKEN` command.

Application vs. syscall Interface

- Assume application wants to write a 16MB buffer.
- POSIX doesn't require `write()` to accept the full buffer size specified in the syscall. So I/O may end up being chopped into smaller chunks.
- Maybe we end up with 12MB + 4MB I/Os submitted through two `write()` calls.

OK, we can live with 2 I/Os.

Kernel Ingress

- The 16MB buffer is contiguous in the application's virtual address space. In physical memory, however, these buffers are typically scattered throughout the address space as individual 4KB pages.
  - The kernel will iterate over these marked pages to build scatterlists and set up I/Os requests for the storage stack.
- The 16MB buffer is contiguous in the application file address space. On disk, however, the filesystem will fit these blocks into discrete extents depending on where there is free space available.
  - Maybe we'll end up with four 1MB + 8MB + 4MB + 3MB regions on the underlying block device.

4 I/Os? Not exactly ideal, but we'll take it.

Controller Constraints

- A controller typically communicates a set of DMA constraints to the storage stack:
  - Max number of scatterlist elements per I/O
  - Whether adjacent scatterlist elements can be merged
  - Maximum I/O size in bytes
  - Segment boundaries that require the scatterlist element to be split even if it would otherwise fit
- Our original 16MB write would probably be chopped up into 32 individual 512KB I/Os by the block layer (assuming things were aligned).

32 I/Os? Oh, dear!

Device Constraints

- The storage device itself may also indicate a maximum transfer length in the Block Limits VPD.
- So maybe we'll now end up with 64 $\times$ 256KB I/Os due to the device imposing additional constraints.

64 bottles of beer on the wall…

RAID / Device Mapper Constraints

- In addition to all this, the filesystem could be on top of a software RAID device. For instance RAID5 over 3 different drives connected to 3 different controllers with 3 different sets of DMA and transfer length constraints.

My brain hurts…

- However, it's a simple iterative process. Each driver will slice and dice a logical block at a time. And eventually the entire 16MB I/O gets mapped and pushed out to the correct device(s) in smaller chunks.

RAID / Device Mapper Constraints

- Some of the 256KB chunks bound for the same device may even be merged into larger I/Os that are contiguous wrt. the device's LBA space.
- In practice, alignment and filesystem allocators mean that even this simple example wouldn't actually go down the stack as 64×256KB chunks but as a random mishmash of various sizes.

Better stop counting those I/Os…

Block Layer Stacking

It's hard to make predictions, especially about the future (Bohr)

- We don't know the block mappings ahead of time.
  - Filesystems delay allocation and placement decisions until the very last minute.
  - Our software block devices can be arbitrarily stacked. Encryption on top of thin provisioning on top of RAID5 on top of RAID1.
- The translation is not necessarily a simple mathematical function such as doing round-robin in 64K chunks over 4 drives. Can't be pre-calculated.

SCSI Copy Offload

With great power comes great pain and suffering (Anonymous)

- The *1:many* problem for `READ` and `WRITE` is aggravated with copy offload because the source and destination LBA ranges need to be sliced and diced independently. There is no guarantee that a 8KB source LBA range on one device maps squarely to an 8KB destination LBA range.
- There is an *m:n* relationship and we don't know ahead of time what it is. We also don't have a way to record the resulting splits as the I/O goes down the stack and comes back up.
- We once tried to do intersection mappings for both I/O topology and `TRIM/UNMAP`. Both attempts proved unworkable given how the stack is currently implemented.

SCSI Copy Offload

- As a result, we do not support offload if an I/O needs to be split.
- The `EXTENDED COPY` implementation works by sending down a copy-in request and if it reaches the SCSI disk driver without being split, we complete it to the upper layers.
- We then submit the copy-out request. This also needs to traverse the entire stack without being split.
- Only if copy-in and copy-out both reach the SCSI disk driver untouched, the copy offload operation can proceed. Otherwise we'll fall back to a manual library function using `READ` and `WRITE`.

SCSI Copy Offload

- You have probably noticed that this `EXTENDED COPY` implementation mirrors the `TOKEN` semantics commands very closely. Not an accident!
- The main difference is that for `EXTENDED COPY`, the token for the copy-in stage is merely the SCSI disk driver signaling that the I/O arrived without being split. Whereas for token-based, the token is actually requested from the storage device. Otherwise the two modes are identical and are implemented using the same plumbing.

Why isn't this upstream yet?

- Several storage vendors discourage mixing copy offload requests with regular `READ/WRITE` I/O.
- The fact that the operation fails if a copy request ever needs to be split as it traverses the stack has the unfortunate side-effect of preventing copy offload from working in pretty much every common deployment configuration out there.
- The storage stack would effectively need to switch away from the iterative stacking approach. It is hard to justify that development effort.
- Maybe Computational Storage will provide the required momentum to revisit some of these problems?