Open Channel SSD

NAND flash SSDs are widely used as primary storage devices due to their low power consumption and high performance. However, SSD's suffer from unpredictable IO latency, log-on-log problems, and resource underutilization.

BY ARKA SHARMA, AMIT KUMAR, ASHUTOSH SHARMA

Along with the adoption of SSDs, the need for more predictable IO latency is also growing. Traditional SSDs that expose a block interface to the host often fail to meet this requirement. The reason is the way NAND flash works. Typically, within an SSD, flash is divided into chips that consist of dies. A die can execute flash commands (read/write/erase) independently. Dies contain planes which can execute the same flash commands in one shot across multiple planes within the same die. Planes contain blocks which are erase units and blocks contain pages which are read/write units.

The chips can be organized into multiple channels that can independently transfer data in and out between NAND and the flash controller. As it is well known that pages in NAND can't be overwritten, a block must be erased first before its pages can be filled with new data. Blocks have a limited number of times they can be erased. This count is also called the PE(Program/Erase) count, which is different for different types of NAND. As an example, SLC NAND has a PE count of around 100,000, the MLC PE count is somewhere between 1,000 to 3,000, and the TLC PE count range is 100 to 300. Typically, SSDs internally run a Flash Translation Layer(FTL) that implements a log-structured scheme which gives the host an abstraction of in-place updates by invalidating the previous content. FTL's also implement a mapping scheme to facilitate this.

As with any log-structured implementation, fragmented writes occur over time which creates the need for garbage collection (GC) to erase invalidated data and create free blocks. In the case of SSDs, this will require moving valid pages from one block (GC source) to another block (GC destination) and then erasing the source block and marking it free. The entire task is performed transparently to the host which faces the drop in SSD performance as well as the GC operations also affect the lifetime of flash media by writing valid data to GC destination blocks. There are several studies and existing solutions to mitigate this like introducing TRIM/UNMAP which aims to invalidate data from the host in such a way that minimizes the number of pages GC operation must move. Multi-stream SSD is a technique to attempt to store data in such a way that data with similar lifetimes is stored in the same erase block, thereby reducing fragmentation which, in turn, relaxes the GC to some degree. Workload classification is another approach of reducing fragmentation. Open channel SSD(OCSSD) is another approach to increase predictability and better resource utilization by shifting some of the FTL's responsibility to the host. Typically, the responsibilities of an SSD can be classified into following categories, data placement, I/O scheduling, media management, logical to physical(L2P) address translation, and error recovery.

OCSSDs can either transfer all (Fully host-managed Open-Channel SSD (1.2)) or some (Host-driven Open-Channel SSD (2.0)) of the responsibilities to the host. Our work is inspired by LightNVM which is Linux's implementation of open channel SSDs and Linux specifics have been modified to fit in FreeBSD's ecosystem. As in LighNVM, it is observed that a shared model of responsibilities achieves a better balance without stressing the host to a greater extent. We explore a model of OCSSDs where data placement, L2P management, I/O scheduling, and some parts of NAND management are done by the host. Some tasks like error detections and recoveries are still done on the device side. The OCSSD exposes a generic abstracted geometry of the media (NAND), wear-leveling threshold, Read/Write/Erase timings, and write constraints (min/optimal write size).

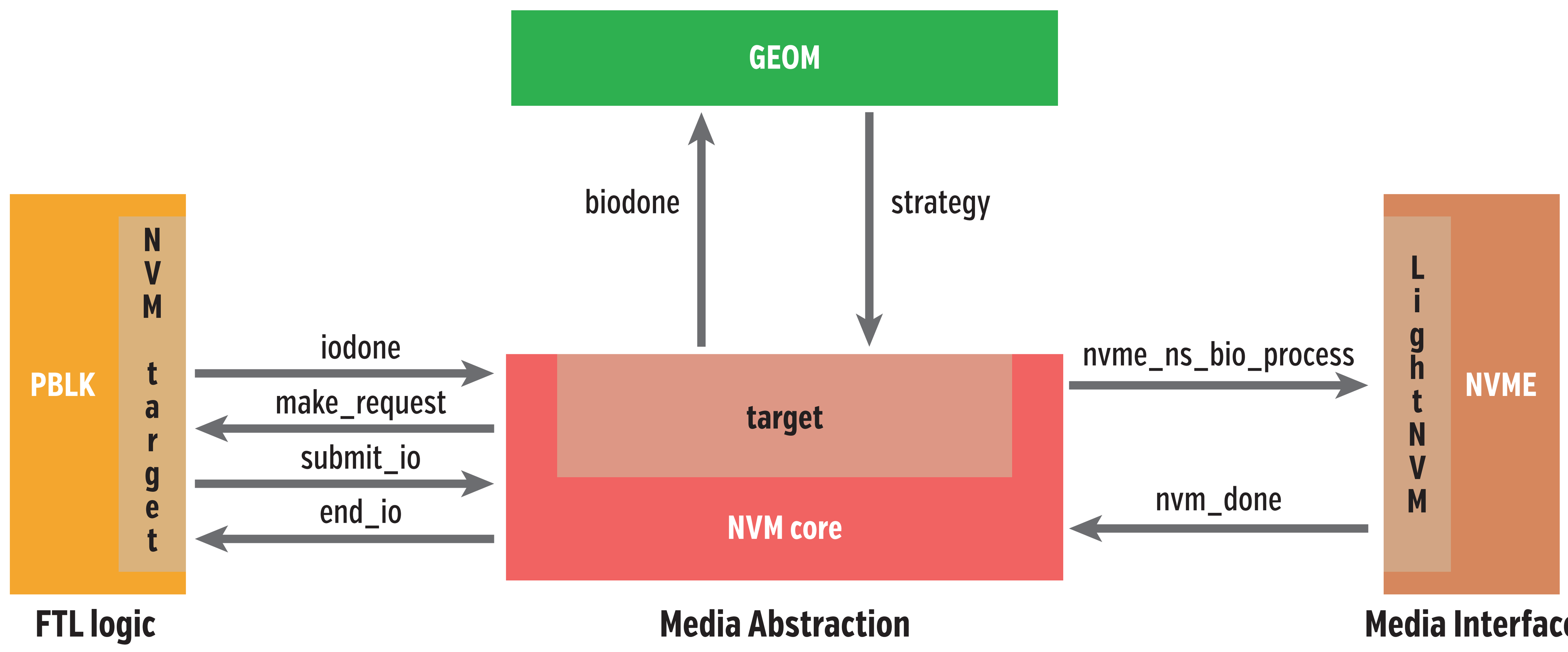
The geometry information typically depicts the parallelism within the underlying NAND media through the number of channels, chips, blocks, and pages. The host can query the state of blocks through commands and get the following information: LBA start address, current write offset within the chunk, and state of blocks (Full, Free, Open, Bad). The drive provides active feedback of chunk health, thus reminding the host to move data from those chunks when required.

Some tasks like error detections and recoveries are still done on the device side.

So far, basic read and write use cases have been tested using FIO. Garbage collection, which is one of the must-have features, hasn't yet been developed due to bandwidth unavailability. All the development efforts have been on QEMU, hence the performance benchmark data is also currently unavailable. Before we received the update about the removal of LightNVM in Linux in 5.15, we planned to on implementing this solution as a GEOM class and with some specific solution where we could consider a custom box with some NVRAM/NVDIMM/PCM as cache and that being coupled with open channel SSDs. But at this point, we have chosen to scrap these ideas. In the future, we look forward to getting involved with work related to NVMe ZNS in FreeBSD.

We have split our work into two components. The FTL part which we call pblk, and the driver which we called lighnvm, keeping the nomenclature similar to LightNVM in Linux. We followed the model of nvd to write the lightnvm driver. The lightnvm driver creates a DEVFS entry "lightnvm/control" which can be used by various tools(nvmecli) to manage the OCSSD device. We have added support for OCSSD devices in nvmecli. The underlying NVMe driver (sys/dev/nvme) initializes the device and notifies the lightnvm driver. The lightnvm driver registers the device to the lightnvm subsystem, the lightnvm system initiates the initialization process and populates the geometry of the underlying media by querying it from the device via the NVMe Geometry admin command(http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf). After the device geometry has been populated, the lightnvm subsystem registers the device along with it's geometry and other NAND attributes.

Once a user initiates the creation of an OCSSD target (via nvmecli), the lightnvm driver carves the requested space out of the OCSSD and creates a "disk" instance for the target which interfaces with geom subsystem. The IOs are intercepted by the strategy routine and forwarded to the pblk subsystem for further processing. The completion of IOs is notified by nvme to the lightnvm, which relays it to the pblk and is subsequently passed up to the geom layer.



We kept the FTL algorithm in the pblk layer largely similar to that of LightNVM. We defined the mapping units to be 4K (also called sectors), which implies that each logical page of size 4K to be mapped with a 4K part of a physical page which is typically larger than 4K. We use nvmecli to carve out the parallel units and create a target. While creating the target, we have an option to choose the target type, which allows us to select the underlying FTL (in case we have more than one) with the target.

As mentioned before, the NAND is divided into chips/dies/plane/blocks. In the context of lightnvm and keeping the terminologies consistent with OCSSD specification, we use the term group for channels, PU or parallel units for chips, and chunk for blocks. OCSSD spec also defines Physical Page Address or PPA which locates a physical page in NAND in terms of group, PU, chunk and page number within chunk. OCSSD compliant devices expose the NAND geometry via the 'geometry' command which is defined in OCSSD specification, and abstracts some of the particularities of underlying NAND media. This allows the user to choose the start and end parallel units which would be part of the target. This also enables the underlying FTL to define 'lines' which is an array of chunks across different parallel units such that the data could be striped to take advantage of the underlying NAND parallelism. This can be achieved two ways: if the target consists of PU's that are connected to different NAND channels, then the data from the SSD controller can be sent to/received from NAND simultaneously. If the PU's of the target are connected to the same channel, the data flow can't happen in parallel. However, once the data flow is complete and flash commands are being executed inside PU's, channels could be utilized for transfer data to/from other PU's. In the case where the target contains one single PU, as expected, we can't have parallelism.

For writing data, we typically write it to a cache and return the success status to geom. We have a writer thread that writes data from this cache to NAND. The size of the cache is computed such that it must accommodate the number of pages that have to be written ahead of a page before data can be read from that page. Suppose the underlying NAND has a restriction that 16 physical pages must be written ahead of a page, and let us say we want to read data from page 10. To be able to reliably read from the chunk, pages up to 26 must be written. Now, if we consider striping, it will take more time to fill those pages, as all the chunks in the line will have same restriction. Also, we must ensure that the maximum number of sectors in a chunk that can be written in a single vector write commands to be fit in cache. The reason for this is that chunks can have program failure and to do a chunk replacement and retry the write command, we need to hold that much data in the cache. And the cache must be able to hold that much data multiplied by the number of PU's in the target. So, to avoid data loss, we need to ensure these pages fit in the cache. The L2P mapping data is maintained in three places: in the host memory which maps the entire target, at the end of the line which maps only the pages written in that line, and in the spare area of the physical page which contains the data of the logical page. As mentioned before, garbage collection has not yet been implemented due to bandwidth unavailability.

As mentioned above, we have a writer thread that reads the data from the cache and writes it to a NAND device. As we defined the mapping unit of the device to be of 4K size, we have divided the cache and the ring buffer in terms of entries with each entry corresponding to 4K of user data. We store some counters in a ring buffer which act as pointers to dictate the writer thread to pick the right ring buffer entry for flushing the data to the NAND device, acknowledging that flush is successful, and updating the L2P map so that the logical page maps to a physical page instead of to the cache entry. These counters store the cache information such as size of the cache in terms of ring buffer entries (4K), how many writable/free entries are available in the cache, how many entries are yet to be submitted to the NAND device, entries whose acknowledgment is yet to be received from the device, entries whose acknowledgment we got from the device, and entries whose physical mapping needs to be updated from cache address to the device's PPA. So, now with the help of these counters, the writer thread will calculate the ring buffer entries whose data need to be flushed to the device. Now it will check if the number of entries (which need to be flushed to the device) is greater than the minimum write pages data (a.k.a. Optimal Write Size). Let's consider Optimal Write Size as 8 sectors (8 * 4K). So, if the number of entries is less than 8, then the thread will come out and retry in the next run. But if the number of entries is greater than or equal to the 8 (Optimal Write Size), then it will read those entries from the cache. While forming the vectored write command to write data to the physical page, we create a meta-area for each page where we write the LBA of the associated page. This is done so that we can recover the mapping in case of power failure. In the current implementation, we have only one active write end, which means we will write to one single line until it is full or there is a program failure, in which case we allocate a new line and write in that. Once we have all 8 (Optimal Write Size) sectors available in the memory pages (data + meta), we will write the data to the device and update the WP (write pointer) of the device and internally in the NAND pages the LBA information will be updated in the spare area. In the case where a write request gets failed by the device, then we will add those failed IOs to a resubmit queue. Here also, the consumer of the resubmit queue is the writer thread. This time, the writer thread will read only those failed entries from the ring buffer (cache). So, now if the number of entries is less than 8 (Optimal Write Size), then we will add padding (dummy pages) and resubmit the write request to the device.

The L2P mapping data is maintained in three places: in the host memory which maps the entire target, at the end of the line which maps only the pages written in that line, and in the spare area of the physical page which contains the data of the logical page.

For the read request we receive the number of sectors requested to be read, along with the starting sector and the data buffer, encapsulated in a bio structure. Consider a read request for 8 sectors. Now, we read the L2P mapping of the first sector. If the logical address of first requested sector is mapped to cache i.e., the data resides in the cache/ring buffer, then we calculate the number of contiguous sectors whose data reside in the cache. Suppose the logical address of all 8 sectors is mapped to cache. Then we just copy the data of all 8 sectors from the cache to the pages of the read bio structure and call the bio_done to send data back to the above layer (geom).

In another scenario, where the first requested sector is mapped with the device, we calculate the number of contiguous sectors whose data reside in the device and we create a child bio for those contiguous sectors and send a read request to the device with appropiate PPA. Now suppose the logical address of all 8 sectors is mapped to the NAND device. Then we will create a child bio of 8 pages and send the read request for those 8 sectors to the device. Meanwhile, the parent (read) bio will wait until we receive the acknowledgment from the device for the read completion. After this, the read bio which was sent from GEOM will, update it's buffer with the data read in child bio, and call the bio_done to send the data back to geom.

In another scenario, where the first requested sector is mapped with the device, we calculate the number of contiguous sectors whose data reside in the device and we create a child bio for those contiguous sectors and send a read request to the device with appropiate PPA.

Now there is another hybrid case, where partial data resides in the device and the remaining data in the cache. Let's consider an example where the first two sectors are residing on the device, the third and fourth sectors are on the cache, and the remaining four sectors again reside on the device. Now, the first step is the same i.e., we find the mapping of the first sector is on the device, we find the contiguous sector count as 2. We create the child bio of two pages, we send the read request to the device using the child bio. Now, we'll find the logical address mapping of the third sector is on cache and once again we get the contiguous sectors count as 2. So, we read the two appropriate ring buffer entries and copy their data to the read (parent) bio's pages. Once again, we find the mapping of the fifth sector is on the device and the contiguous sectors count is 4. This time we create another child bio to read the remaining four sectors from the device. Now the parent (read) bio must wait until we receive the acknowledgment from the device for both child BIOs. In the end, read IO will get the data from both child BIOs and the cache, and then we call the bio_done and complete the read request.

ARKA SHARMA has working experience on various storage components like drivers, FTLs, and option ROMs. Before getting into FreeBSD in 2019, he worked in WDM mini-port and UEFI drivers.

AMIT KUMAR is a system software developer and currently works on storage products based on FreeBSD. He has been a FreeBSD user since 2019. In his spare time, he likes to explore the FreeBSD IO stack.

ASHUTOSH SHARMA currently works as a software engineer at Isilon. His main area of interest is storage subsystems. In the past, he worked on Linux md-raid.