Hexa: Overcoming Capacitance Constraints in SSDs

Anonymous

***Abstract*—The growth in SSD capacity is reaching its limit due to the stunted growth of capacitors—electrical components that store charge to protect data for the volatile memory in case of power loss. This paper presents `Hexa`, a novel SSD-internal DRAM management scheme that allows the SSD capacity to scale beyond the slow growth of capacitors. `Hexa` suppresses an increase of the dirty memory footprint within buffer using the deep queues available in today's storage interfaces. We implement our design in `FEMU`, an open-source SSD development framework and demonstrate that `Hexa` delivers IOPS close to 90% of performance at only 1% of capacitance compared to the existing scheme.**

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The enterprise-class SSDs adopt the capacitor to protect data durability in case of power crash. This technique is called Power-Loss Protection (PLP) [11, 18, 23] and it is needed because SSDs use a DRAM as an internal buffer for absorbing user writes and caching translation information (also known as mapping table). If they are not protected, SSDs will have not only a data loss and/or corruption but also a long recovery time to build an up-to-date mapping table by scanning entire flash drives. To preclude this situation, the enterprise-class SSDs rely on the capacitors that reserves energy to safely persist data of the volatile buffer in a power loss.

However, the heavy reliance on capacitors is no longer sustainable as the increase in SSD far outpaces the increase in capacitor density. The SSD has increased significantly in density for the past decade. In 2011, a typical 2.5-inch SSD had 256GB capacity, but by 2018, a high-capacity SSD boasted a 30TB, expanding by 100× over the past ten years [1, 21]. This remarkable growth of the device-capacity is thanks to the advanced scaling technologies such as nanoscale fabrication [3] and multi-layer stacking [20]. Al(aluminum) and Ta(tantalum)-electrolytic capacitors used in SSDs have increased in density by tenfold from 1960 to 2005. This is approximately 50x slower than the SSD density increase rate. Given that the internal buffer size increases in proportion to the storage capacity, the slow scaling of capacitors will eventually limit the amount of DRAM that can be used in an SSD. This, in turn, will also limit the storage capacity as the size of DRAM and aggregate flash capacity proportionally scale [19, 22].

This paper presents `Hexa`, a novel SSD-internal DRAM management scheme that allows the SSD capacity to scale beyond the slow growth of capacitors. SSD-internal DRAM is used for (1) caching translation information (also known as mapping table) and (2) buffering user writes. In typical SSD designs, most of the DRAM is used for caching the mapping table and and the buffer for user writes is kept at a minimal (just enough to hide the flash program latency) [13]. As an example, Samsung PM1633a 15.36TB SSD houses 16GB DRAM [1]. Given that the mapping table size is typically 0.1% of storage capacity [19, 22], we can assume that only 4% of DRAM is used for a write buffer.

As opposed to a memory pressure, the negative impact of data loss is equally serious for both data. Because an SSD writes the associated LPN (Logical Page Number) in the OOB (Out-of-band) area of the physical page, it is virtually possible to recover the up-to-date mapping table by scanning the entire NAND flash memory. However, because it takes prohibitively long, particularly for the scalable SSDs, PLP-SSD snapshots an entire mapping table into the specific area in NAND flash in a power loss and loads it into DRAM at a reboot. The user data also offers no alternative but for PLP as it cannot be recovered after a crash. For the PLP-SSD, the host system ensures reliability assuming that all acknowledged data survive a power outage, and thus, the loss of user data can lead to a catastrophic result.

With these properties in mind, we invent an in-device buffer management mechanism for SSD under capacitance constraints. `Hexa` partially protects the mapping table, while fully protecting user data under capacitance constraints. If the number of dirty pages in mapping table goes beyond the limit, changes are immediately flushed to NAND flash. Instead, `Hexa` buffers more user writes so that mapping entry eviction becomes more efficient by aggregating dirty updates. This substantially reduces the amount of mapping table-related write traffic, and in turn, improves the overall performance under capacitance constraints.

`Hexa` is built upon the current trend of increasing the queue depth of the storage interfaces. SATA and SAS support a single queue with 32 and 245 commands, but NVMe has up to 65,535 queues with as many as 65,536 commands per queue. This extension allows SSDs to further optimize the internal activities by taking advantage of the outstanding request information.

We implement `Hexa` in `FEMU`, an open-source SSD development framework [17]. The performance evaluation with

SSD Model	Manufacturer	Class	PLP	Capacitor
950Pro, 850Pro	Samsung	Client	None	-
M500	Micron	Client	Partial	Ceramic
M500DC	Micron	Enterprise	Full	Tantalum
PM863, SM863	Samsung	Enterprise	Full	Tantalum
DC1000B	Kingston	Enterprise	Full	Tantalum
DC S3700, S3500	Intel	Enterprise	Full	Aluminum

TABLE I: **Power Loss Protection in SSDs [11, 18, 23]**. EJ: This table is also included in ISLPED paper. Should we take it out?

various workloads shows that Hexa offers 82% and 94% of IOPS of the full-protection SSD when a protected ratio is 1% and 10%, while a conventional SSD provides 69% and 81% of performance.

II. Related work

The need for reducing the energy consumption needed for power-loss protection arises in different contexts. A few studies reduce the total energy consumption by speeding up the back-up process at a power failure using the fast media. Guo et al. reduce the capacitance requirement by writing back the volatile buffer data into PRAM (Phase Change Random Access Memory), which is faster and uses lower power than NAND flash [7]. They argue that this reduction enables to replace the supercapacitors that are suffering from serious aging problems with the regular capacitors, which have more reliable characteristics [9]. This consequently enhances the robustness of storage device. As a similar approach, Smartbackup [10] proposes dynamic NAND channel allocation and SLC (single-level cell) mode programs to make the dump process shorter at sudden power-off. It makes full use of available SSD channels and dynamically adjusts these channels based on the available power of the capacitor to exploit the nature of high parallelism on NAND flash arrays. In addition, as the SLC mode program shows significantly shorter time than subsequent MLC, TLC, or QLC mode, it programs the target page to dump in SLC mode to achieve shorter time required for dumping process.

Another approach to reducing the capacitor size is protecting a part of the volatile buffer. DRWB (Dual-Region Write Buffer) divides the internal-SSD buffer into small protected region (backed by a capacitor) and large unprotected region and when the data on unprotected region is updated, the delta for the page is logged in the protected region [14]. With this differential logging, DRWB logically realizes the non-volatile buffer using a small size of capacitor. However, the proposed technique only regards the user data, having no consideration on the metadata such as mapping table, despite that it actually accounts for most of the internal buffer of SSDs. Furthermore, commercial SSDs typically do not cache read data in the buffer because the host memory can serve as a cache memory of the storage device. For these reasons, the effectiveness of DRWB may be limited in practical environment.

Some studies explore ways of using the internal write buffer efficiently in scalable SSDs. Chen et al. project that even the high capacity of SSDs will use the small size of write buffer because the capacitor that protects the buffer does not scale well due to the cost, size, and reliability constraints [4]. Nevertheless, they observe that the small sized write buffer can be effective for reducing write traffic in particular applications that perform journaling heavily. Motivated by this observation, they present the application-SSD co-design to reduce the data writes buffered for heavy logging/journaling applications. They propose to protect write-hot log/journal data with capacitors while the log/journal data being durable. In addition, they propose NVMe interface extension for host to notify SSDs the ranges of write-hot LBAs for more efficient

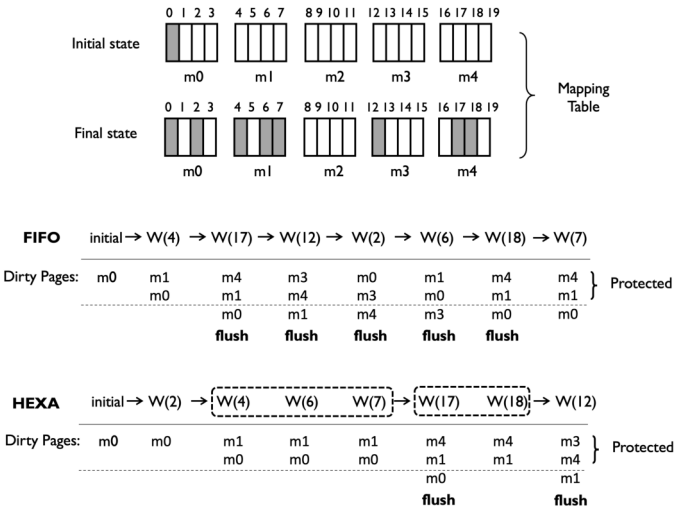


Fig. 1: **Hexa-SSD buffer management scheme**

protection by capacitors with reduced complexity of hot/cold separation. It reduced substantial amount of flash memory write traffic with few megabytes of capacitor-powered write buffer, but it is specific to heavy log/journal applications and requires change of application code to benefit from its scheme.

SpartanSSD [16], which is most related with our work, pinpoints capacitance constraints in scalable SSDs and reduces capacitance requirements by making use of elastic journaling. Spartan-SSD logs the mapping information updates into the in-device journal so that the writes to the mapping table can be buffered. They use a hybrid journal that is backed by a small size of DRAM and flash memory. This hybrid journal is highly flexible in terms of capacity, and thus, it enables a timely checkpoint that reflects a log data to the mapping table and flushes dirty map pages into NAND flash chips. Although Spartan-SSD also reduces translation-related writes under capacitance constratins, it not only have double write for maping information updates, but also it essentially increases a recovery time, which could be highly harmful when the multiple SSDs are running simultaneously.

III. Design

Hexa partially protects the mapping table with limited capacitance. When the dirty pages of mapping table become more than the maximum number of protected pages, Hexa flushes them to flash memory based on the LRU (Least-recently Used) algorithm. Because this flush operation does not arise with SSD using PLP, mitigating the effect of this overhead is a key strategy to achieving high performance under capacitance constraints. To this end, Hexa presents a cost-effective scheduling scheme for the in-storage buffer. Hexa prefers to force the user data that increases the dirtiness of the mapping table the least to flash memory. This scheme reduces the dirty page footprint of the mapping table at a time window by enhancing the locality of updates. As a result, the frequency of flush operation for the mapping table can be largely reduced.

Figure 1 compares the flush overhead of FIFO and Hexa scheduling in SSD buffer. In this example, there are seven write requests in the device queue, sent from host in the following order: W(4), W(17), W(12), W(2), W(6), W(18), and W(7). The mapping table has one dirty page (m0) at an initial state. We assume that 2 out of 5 pages of the mapping table are protected. FIFO writes the user data in the buffer to flash memory in arrival order. With this scheme, the mapping table would be randomly updated, generating a large number of dirty pages at a time window. Consequently, FIFO incurs a total of five flushes of the mapping table page during the write process.

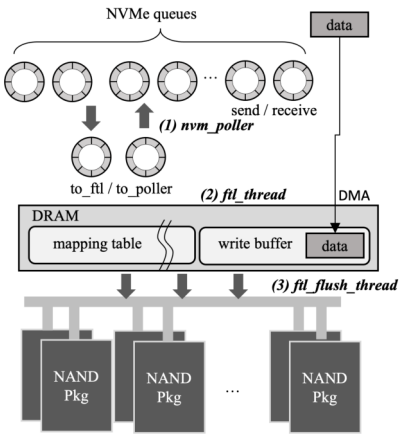
In contrast, Hexa calculates the write cost for each data that indicates an increase in the number of dirty pages of the mapping table when it is flushed, and it processes the request with minimum cost first. In this example, the write request W(2) has a top priority because its associated mapping table page (m0) is already dirty, and thus it does not add the dirty pages of the mapping table. Next, the write requests W(4), W(6), and W(7) are processed. Because their address mapping entries are located in the same page of the mapping table, the cost of flushing them is reduced to one third. With this scheme, Hexa can reduce the footprint of mapping table updates within time intervals, thereby delivering only two flushes of the mapping table for the same task.

IV. Implementation

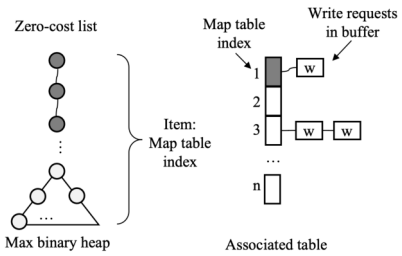
We implement Hexa in FEMU, an open-source SSD development framework [17]. Fig. 2 shows the overall architecture of Hexa-SSD and its internal data structures. As the original version of FEMU directly writes data to flash memories without write buffering, we extend it to use a small-sized write buffer, which aggregates and batches user writes into the underlying flash memory.

Hexa-SSD maintains three different threads that are executing concrrently within SSDs. The nvm_poller takes a charge of transferring requests between NVMe queues and FTL-internal queues. The FTL-internal queue consists of a pair of sub-queues, each of which is named to_ftl and to_poller). This separation is intended to enable a non-blocking access to queues by allowing only a single writer for each queue. Second, the ftl_thread essentially handles the ingress requests from the internal queues. For write, it transfers data from the host memory to the SSD-internal write buffer with DMA and updates the associated entry in a translation page to point to the write buffer. Then, it notifies the completion of request to the nvm_poller by enqueueing the acknowledgement into the to_poller queue. Because Hexa protects the entire space of write buffer with capacitance, data persistency is guaranteed for all acknowledged writes. For read, the ftl_thread retrieves the requested data by consulting the mapping table and transfers it to the host.

The ftl_flush_thread plays a role of writing data from a DRAM-buffer into a flash memory. With the FIFO policy, the user writes are issued to NAND flash memory in the order they arrive into the buffer. However, Hexa flushes



(a) Architecture



(b) Data structures for FTL

Fig. 2: **Hexa-SSD** Internals.

buffered writes in the order such that it least increases the dirty memory footprint of the mapping table. To realize this design, Hexa maintains two data structures, as depicted in Fig. 2(b). First, *a zero-cost list* that holds the indexes to tranlsation pages that is already in a dirty state, and second, *a max binary heap* that maintains the indexes to translation pages sorted by the number of buffered user write requests associated with that page.

When a half of the write buffer becomes occupied, flushing is invoked. Hexa-SSD first flushes user data whose translation pages in the zero-cost list, and then persists user data as their translation pages are ordered by the max binary heap. By doing so, each user write minimizes the number of eventual translation page write, and each translation page write maximizes the number of persisted mapping entries. These data structures are updated by the ftl_thread when a write request arrives at SSD. To exploit the SSD internal parallelism, we send data to flash memory in batches by the number of NAND flash chips that can be written simultaneously.

Once the write operations of NAND flash memory complete, ftl_flush_thread updates the mapping table entries to point to the physical address of the data in a flash memory. At this moment, if the number of dirty mapping table pages goes beyond the protectable number of pages, ftl_flush_thread persists the mapping table page to flash memory. This is also conducted in batches by the number