Overcoming Capacitance Constraints in SSD

Sanghyun Nam
Soongsil University

Seungmin Shin
Soongsil University

Sungjin Lee
DGIST

Bryan S. Kim
Syracuse University

Eunji Lee
Soongsil University

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 `Dawid`, a novel SSD-internal DRAM management scheme that allows the SSD capacity to scale beyond the slow growth of capacitors. `Dawid` 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 `Dawid` delivers IOPS close to 90% of performance at only 1% of capacitance compared to the existing scheme.

1 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) 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, 7]. This remarkable growth of the device-capacity is thanks to the advanced scaling technologies such as nanoscale fabrication [**?**] and multi-layer stacking [**?**]. 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 [6, 8].

This paper presents `Dawid`, 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 capacitance is used for protecting the mapping table (to keep as many translation entries in DRAM) and the buffer for user writes is kept at a minimal (just enough to hide the flash program latency) [4]. As an example, Samsung PM1643 30.72TB and PM1633a 15.36TB house 40GB and 16GB DRAM, respectively [1]. (typically 0.1% of storage capacity [6, 8])

However, in our design, we take a radically different approach. We buffer 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.

The data maintained in the buffer can be classified into two types: the actual user data and the metadata for SSD management (i.e, mapping table). When the buffer is partially protected, the number of dirty pages is limited to the maximum amount of data that the on-board capacitance can protect. If the number of dirty pages goes beyond the limit, changes should be flushed to the flash memory immediately to meet the durability constraint for SSDs.

2 Related work

This paper builds upon several prior work done on SSDs. In this section, we briefly outline them and discuss our work in relation to them.

2.1 Mapping Table

The large memory footprint of the mapping table has been a drawback for SSDs. Several works attempted to reduce the memory usage of the mapping table. D-FTL performs on-demand paging for the mapping table using DRAM as a translation cache [**?**]. It maintains the entire mapping table in flash memory, caching a subset of the translation information in the limited size of DRAM. This scheme trade-offs the performance with the memory usage for high-capacity SSDs and it can deliver a poor performance under the workloads with weak locality.

HP-FTL saves the internal memory space by adopting the hash-based mapping. In the page-level FTL, each entry of the mapping table is typically 4byte-long which represents the PPA (Physical Page Address) for the LPA (Logical Page Address) [6]. HP-FTL determines the physical block to place the incoming data using a hash function. To reduce a hash-collision, HP-FTL use multiple hash functions and each entry of the mapping table maintains the hash function ID effectively used and the physical page offset within the block. As an example, when 4 hash functions are used and a block consists of 64 pages, each entry requires 8-bit only. This size is one fourth of the original one. However, this approach significantly increases the GC(Garbage-Collection) overhead because it randomly scatters writes onto the contiguous LPNs. In addition, the fully associative secondary mapping table, which is used to resolve the hash collision, offsets the benefit of the hash-based mapping by maintaining both LPA and PPA information in the table. These works are orthogonal with capacitance-saving techniques that care about the persistence management of the buffered data within DRAM.

S-FTL proposed a method for efficiently caching the mapping table in a small amount of DRAM cache [**?**]. S-FTL uses an extent-like data structure to reduce the in-cache translation memory footprint. It also holds the translation updates in DRAM to reduce the write-back traffic associated with the mapping table. When a dirty mapping page needs to be evicted, S-FTL flushes the mapping table page into flash memory when the number of dirty entries of the page goes beyond the threshold; otherwise, it evicts the page without flushing, keeping the dirty entries within DRAM. The dirty entries are reflected to the mapping table page when it is loaded later.

SHRD [**?**] studied the sequentializing scheme to reduce the write-back overhead of the demand-loading mapping table. They argue that the random writes make dirty entries sparsely dispersed among the mapping pages, decreasing the cache hit ratio of mapping table significantly. To relieve this problem, they transform the random writes into the sequential pattern by maintaining a log buffer in SSD. That is, the random writes are temporarily redirected into the log buffer mapped with sequential LPN(Logical Page Number)s and the redirection information is maintained in a device driver to service the read requests correctly. The mapping table entries for the original LPNs of buffered writes are updated in the order of the original LPN when the log buffer is exhausted. This scheme improves the spatial locality of mapping entry updates, achieving the higher performance of SSD when the demand-loading mapping table is in use.

S-FTL and SHRD have a similarity with our policy in that they also attempt to aggregate the dirty entry updates for a mapping table to reduce a write traffic associated with a mapping table. However, they essentially tackle the write-back cost of the cached mapping table, not considering capacitance constraints.

2.2 Capacitance Constraints

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 [**?**]. 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 [**?**]. This consequently enhances the robustness of storage device. As a similar approach, Smart-backup [**?**] 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 [**?**]. 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 [**?**]. 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 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 [**?**]

3 Design

This section describes the design and implementation of `Dawid` buffer in detail. § 3.1 overviews the overall architecture of the in-storage buffer under capacitance constraints, and § 3.2 presents the I/O scheduling algorithm to reduce the write traffic in `Dawid`.

3.1 Overview

To satisfy a high demand for storage capacity in modern applications, the SSDs have become highly scalable with advances in density-increasing techniques. The state-of-the-art SSDs provide tens of TBs capacity and this trend is expected to continue in the future. The problem is that the capacitors, an integral component used in SSDs to protect the buffer data in power outage, is unable to keep up with the significant density increase speed of NAND flash memory. The SSD-internal buffer is typically 0.1% of the storage capacity in size. To protect the entire buffer of SSD, the capacitors should have been improved at the same pace with SSD in density; its density has enhanced only at one-fifth speed of SSD density improvement. With this density gap, the PLP with full protection is no longer feasible in SSDs; it not only severely limits the form factor of SSDs requiring deployment of a large number of capacitors but also significantly increases the manufacturing cost of SSDs.

`Dawid` is designed to efficiently maintain the in-storage buffer under capacitance limitations. Table **??** shows the breakdown of the in-storage buffer usage for each component for 512GB SSD that has an architecture shown in Table **??**. The user data buffer is employed to fully exploit the underlying flash parallelism. Hence, it is typically twice the size of all pages that can be programmed in parallel, which is 4MB in this setting, while it may vary depending on the design choice. The mapping table that translates LPN(logical page number) to PPN(physical page number) accounts for 512MB, which corresponds to 97% of the buffer size. Other metadata including mapping table directory uses a total of 10.5MB buffer.

To overcome capacitance constraints for SSD, `Dawid` sacrifices on some durability of mapping table, while protecting the user data and the metadata other than mapping table with capacitors. The user data persistence should be synchronously guaranteed with the host request to conserve the properties of existing SSDs with PLP. For this reason, making a compromise on it can lead to a serious performance penalty in SSD. On contrary, the requirements for the mapping table update are less stringent; it does not have to be immediate upon a host request because the address translation is necessary only when the associated data is actually programmed to the flash memory. This nature allows a room for reasonable trade-off between capacitance and performance, by effectively maintaining the persisting overhead of mapping table under capacitance constraint. Other metadata updates are also asynchronous with the host request, but they use only a marginal space of the buffer; sophisticating their management mechanism for further capacitance saving is cost-ineffective.

3.2 Least Increase of Dirtiness Scheduling

`Dawid` partially protects the mapping table with limited capacitance. When the dirty pages of mapping table become more than the maximum number of protected pages, `Dawid` 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, `Dawid` presents a cost-effective scheduling scheme for the in-storage buffer, called LIDF (Least Increase of Dirtiness First). LIDF 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 LIDF scheduling in `Dawid` 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

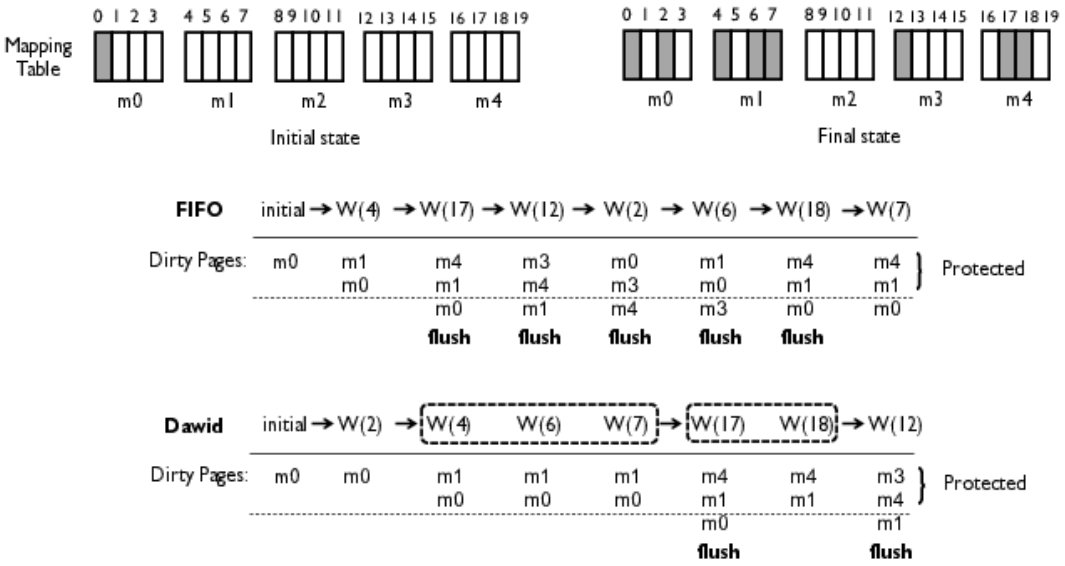


Figure 1: **Dawid Buffer Overview.**

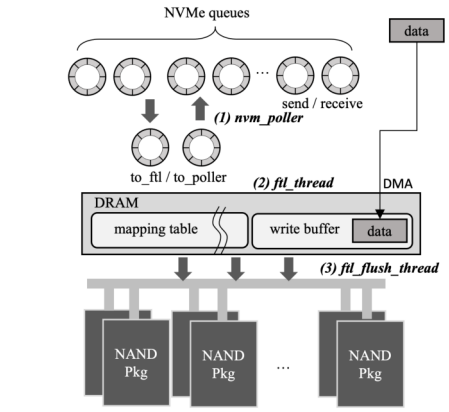
of dirty pages at a time window. Consequently, FIFO incurs a total of five flushes of the mapping table page during the write process.

In contrast, LIDF 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 policy, LIDF can reduce the footprint of mapping table updates within time intervals, thereby delivering only two flushes of the mapping table for the same task.

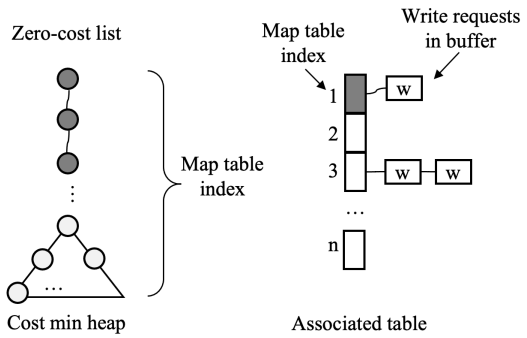
4 Implementation

We implement `Dawid` in `FEMU`, an open-source SSD development framework [5]. Fig. 2 shows the overall architecture of `Dawid`-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.

`Dawid`-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-



(a) Architecture



(b) Data structures for FTL

Figure 2: **Dawid-SSD**

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 `Dawid` 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, `Dawid` flushes buffered writes in the order such that it least increases the dirty memory footprint of the mapping table. To realize this design, `Dawid` 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. `Dawid`-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 of NAND flash chips that can be written simultaneously.

5 Evaluation

We measured the performance of `Dawid` using fio benchmark [2], running 4KB sequential writes, 4KB random writes, and the skewed read-write mixed workload that follows JESD219 using 8 threads. A total of 90GB of data was written to the 30GB area.

6 Conclusion

In this paper, we present the design, implementation, and evaluation of JellyFish, a skip list that reduces the performance overheads for MVCC. JellyFish separates per-key updates from the key indexing, allowing fast list traversals, efficient point queries, and fewer pointer changes for updates. The benefits of our design are amplified as the memory size increases, and our evaluation results show that JellyFish not only improves the throughput, but also the latency significantly.

References

[1] AnandTech. Samsung 30.72 TB SSDs: Mass production of PM1643 begins. https://www.anandtech.com/show/12448/samsung-begins-mas 2018.

[2] Jens Axboe. fio - flexible i/o tester. https://github.com/axboe/fio, 2021.

[3] Jens Both. The modern era of aluminum electrolytic capacitors. *IEEE Electrical Insulation Magazine*, 31(4):24–34, 2015.

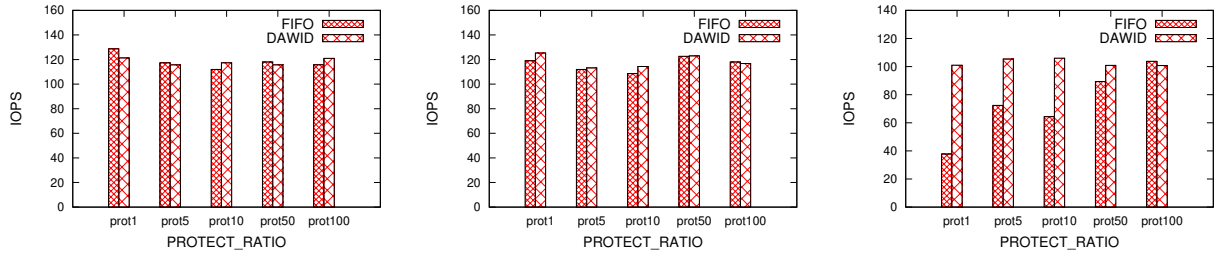
[4] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. Durable write cache in flash memory SSD for relational and nosql databases. In *ACM International Conference on Management of Data (SIGMOD)*, pages 529–540, 2014. https://doi.org/10.1145/2588555.2595632.

[5] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. The CASE of FEMU: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST)*, pages 83–90, 2018.

[6] Fan Ni, Chunyi Liu, Yang Wang, Chengzhong Xu, Xiao Zhang, and Song Jiang. A hash-based space-efficient page-level FTL for large-capacity SSDs. In *2017 International Conference on Networking, Architecture, and Storage (NAS)*, pages 1–6. IEEE, 2017.

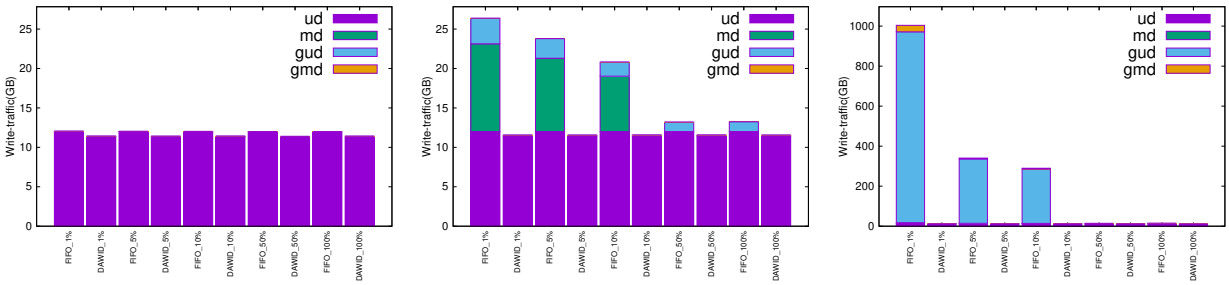
[7] Samsung. Samsung SSD 830 series. https://www.samsung.com/us/support/owners/product/128gb- 2011.

[8] Samsung. Samsung V-NAND SSD 860 QVO. https://www.samsung.com/semiconductor/global.semi.static/ 2013.



(a) Sequential (b) Random (c) JESD

Figure 3: **IOPS**



(a) Sequential (b) Random (c) JESD

Figure 4: **Write Traffic**