Programming an SSD Controller to Support Batched Writes for Variable-Size Pages

Jaeyoung Do
Microsoft Research
Redmond, WA, USA
jaedo@microsoft.com

Chen Luo*
University of California, Irvine
Irvine, CA, USA
cluo8@uci.edu

David Lomet#
Microsoft Research
Redmond, WA, USA
lomet@microsoft.com

***Abstract*—Exploiting a storage hierarchy is critical to cost-effective data management. However, most systems are challenged when data is not in cache because of the additional I/O to move data between SSD and main memory. To improve both cost and performance, some systems use a log structured store to write a batch of pages instead of a "block-at-a-time". However, host-based log structuring incurs the additional cost and complexity of garbage collection and recovery, duplicating similar SSD FTL functionality. In prior work, we presented a customized SSD controller implementation for an Open-Channel SSD to enable host computers to write batches of fixed size pages. This current work is a major redesign to support a batched write interface with variable size pages. Variable size pages can enable easy support of data compression and encryption, as well as reducing internal page storage fragmentation, e.g, within a B-tree. Thus it further improves I/O performance while making it easier and more efficient to support these capabilities.**

I. Introduction

A. Improving Cost/Performance

I/O operations per second (IOPS) can be a limiting factor in systems performance. SSDs have improved dramatically in the IOPS that they can support. Nonetheless, each I/O operation produces an execution burden on the host system. A traditional block-at-a-time (BAAT) SSD interface incurs the host I/O execution path cost for each block (page) written. An SSD that supports a batch (of pages) at a time interface incurs that I/O execution path cost for each batch, hence amortising the execution cost over many pages.

Log structured stores [10], [11] do batching using a conventional BAAT interface. Log structuring never updates in place. Hence, it incurs additional costs to enable batching. (1) The latest location where the page has been written must be durable across system crashes. (2) Old versions of pages need to be garbage collected to avoid their filling up the SSD. While log structuring has a positive cost and performance advantage versus simple BAAT approaches, some of this advantage is given back to deal with durability and garbage collection (GC). And log structuring adds to host implementation complexity.

Modern SSDs also implement log structuring to support a flash translation layer (FTL) in their controllers. The FTL has the same durability and GC requirements as a log structured store. Hence, host-based log structuring overlaps with the functionality of modern SSD controllers and consumes CPU and I/O resources for this duplicate functionality. It is highly desirable to eliminate this duplication by removing log structuring from the host with the SSD controller providing batch at a time capability. This eliminates CPU GC and durability overhead since these functions can now be performed completely in the SSD controller.

Our purpose is to improve the cost/performance of the data management layer by using the re-programmed SSD to reduce I/O costs. While our in-house data management system is the Bw-tree key-value store [9] with its log structured store [10], supporting a batch I/O interface by pushing log structuring into the SSD controller enables many disk-based data management systems (data caching systems [4]) to improve their cost/performance. This can be seen in Figure 1 where executing an operation on in-memory data is plotted versus executing an operation on data that resides on an SSD. SSD storage cost is lower, while operating cost is higer due to the cost of I/O needed to bring the data into main memory. Figure 1 shows how lowering the cost of I/O can lower cost over a large part of the performance range.

B. Variable Size Pages

We previously implemented an SSD controller supporting a batched write interface with fixed size pages [8]. The current paper describes ELEOS, an SSD controller implementation that supports a batched write interface with variable size pages. Compared with our prior fixed size page implementation, supporting variable size pages required a major re-design of garbage collection, provisioning, and page management across the I/O interface. In addition, we substantially improved the performance of checkpoint and recovery.

Why are variable size pages potentially important? Internal fragmentation (unused storage resulting from mapping variable length data to fixed size containers) can be reduced using variable size pages. For example, B-tree pages generated in the usual way, have about 70% storage utilization. When a full page splits, the data is divided into two pages, each about half full, then the process repeats. Variable size pages can squeeze out that internal fragmentation and reclaim the storage. The Bw-tree [9] supports variable size pages, enabling its pages to approach 100% storage utilization. Storing variable length

* Work performed during internship at Microsoft Research.
# Retired from Microsoft Research in October, 2020.

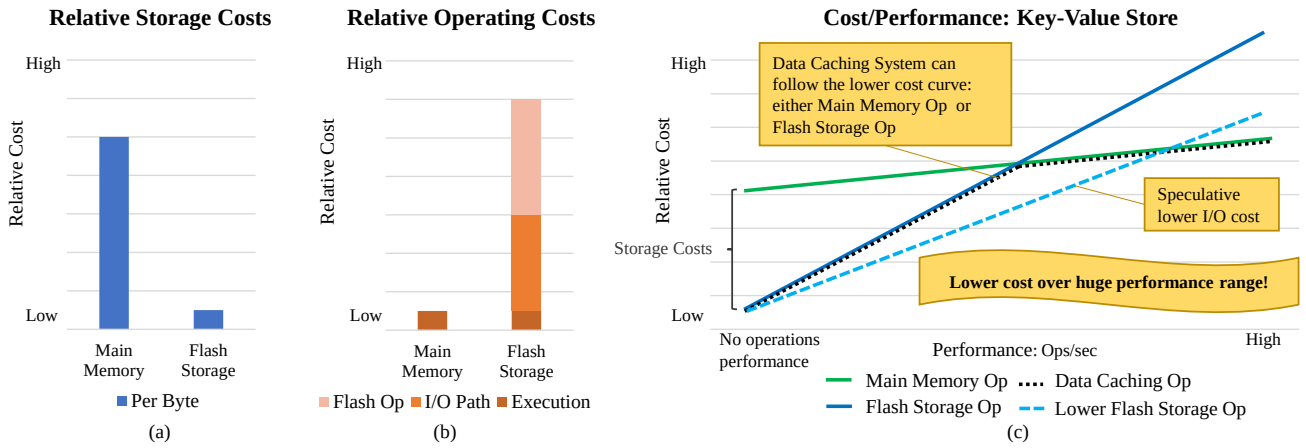

Fig. 1: (a) flash storage cost is lower, but (b) execution cost is higher for reading the data into main memory. (c) Cost vs performance for key-value store operations, showing cost when data is in main memory and when it comes from an SSD. The dotted light blue line shows how costs are reduced if I/O cost can be reduced.

blobs is another case where having variable size pages reduces the internal fragmentation that results from mapping data to fixed size pages.

Even when starting with fixed size data, desired operations on it can produce variable length results. Both encryption and compression usually change the length of the data involved. Thus, we expect compressed data to be shorter than the original starting data. Similarly, data encryption usually increases the length of the data. While current systems work around this change in data length, there is usually a system complexity cost to be paid, and sometimes an added execution cost as well. Direct support for variable size pages is a major simplification for a data management system when dealing with these functionalities.

C. This Paper and Our Contributions

The rest of the paper is organized as follows. Section II provides necessary background for this work. Section III presents an architectural overview of ELEOS. This architecture and its semantics is one of our contributions. Sections IV and V describe how writes and reads are supported in ELEOS respectively. Our write path implementation needs to handle variable size pages, which poses a new set of problems. This is our second contribution. Section VI describes the garbage collection in ELEOS with a novel GC algorithm. The GC algorithm, described elsewhere, is a major advance in log structured GC. Section VII describes how write failures are handled in ELEOS. Dealing with these failures is a challenging part of getting an FTL "right". Section VIII discusses the recovery techniques used in ELEOS. Recovery presents a number of unique challenges and constitutes our third contribution. Section IX presents the experimental evaluation of ELEOS. This evaluation is our fourth contribution and demonstrates the value of our batch interface approach. Finally, Section X concludes the paper.

II. Background

A. Log Structuring

Log structuring [11] was originally proposed to minimize random I/O overhead on hard disks. Unlike update-in-place systems that incur many random I/Os to overwrite pages, a log structured system always appends newly written pages using sequential I/Os to write large buffers of pages. This design maximizes system write throughput by exploiting large sequential I/Os, but has two important implications.

- The system must maintain a persistent mapping table that stores the latest location for each page. The mapping table must be durably updated for each write.
- Garbage collection must be performed to reclaim storage space occupied by obsolete versions of pages. Otherwise, the system will soon run out of space.

The performance penalty of random I/Os have been largely eliminated from modern SSDs, though not the cost of executing the write code path. However, due to the special erase-before-write semantics, SSDs, in their controllers, usually implement log structuring to manage the internal storage media, as discussed above.

B. Solid State Drives (SSDs)

An SSD contains two main components [12], namely the flash storage media and the controller. The storage media is a persistent array of multiple flash channels. A flash channel provides communication between the SSD controller and a subset of flash chips, and a chip consists of multiple blocks, each of which holds multiple pages. A major characteristic of flash is that it must be erased before data is written and can endure only a limited number of erases before it can no longer be used. The unit of erasure is what we refer to as an erase block (EBLOCK), while a read is to a read page (RBLOCK), and a write to a write page (WBLOCK) (see Table I). Higher I/O performance from the flash storage media requires writes in parallel to channels and chip level interleaving techniques.

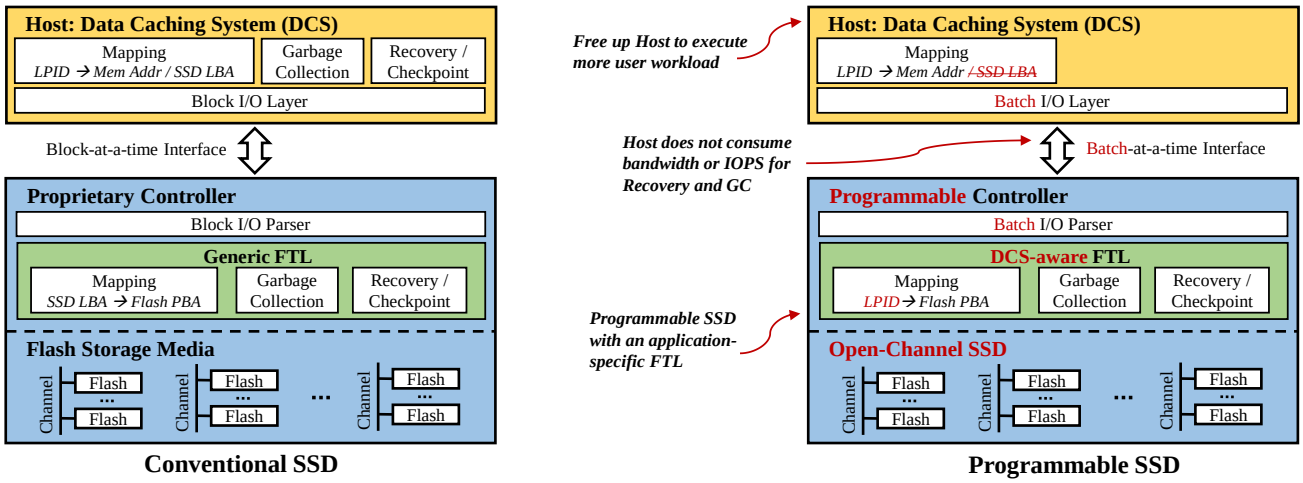

Fig. 2: Left: host-based log structuring built using a conventional SSD. Right: host using an LS engineered controller that allows a batch interface.

Term	Example Size	Description
RBLOCK	4KB	Smallest readable storage unit
WBLOCK	32KB	Smallest writable storage unit
EBLOCK	8MB	Smallest erasable storage unit
TAG	16B/RBLOCK	Controller accessible metadata

TABLE I: Flash memory terms

The SSD controller is commonly implemented as a system-on-a-chip for cost-effective management of the underlying storage media. To deal with the erase-before-write semantics of NAND flash, the controller implements a form of log structuring [11] that supports an FTL mapping logical addresses to physical flash addresses. As in all log structured systems, garbage collection is required to reclaim obsolete pages in flash EBLOCKs.

The SSD controller "firmware" has previously been proprietary with only a block I/O interface exposed to users. However, this leads to suboptimal storage utilization and performance. If we are to overcome this problem, SSD's internal media geometry and parallelism must be exposed so that it becomes possible to control data placement and I/O scheduling in a better way, dependent on user requirements. The industry is undertaking two directions to address this, (1) building customized SSD controllers via programmable SSDs [3], [6], [7], [15] or (2) moving FTL functionalities to the host via Open-Channel SSDs (OCSSD) [13], [14].

III. System Architecture

ELEOS is implemented on top of OX [1], a programming framework for building programmable SSD controllers. ELEOS builds a customized FTL to support batched writes and variable size pages. Fig. 2 compares the architecture of a traditional SSD and that of our SSD controller. As one can see, ELEOS now provides a batched I/O interface for better performance. Moreover, it also eliminates the log-structuring overheads, i.e., garbage collection and recovery, from the host.

A. I/O Interface Semantics

Unlike conventional SSDs, which support a BAAT interface, ELEOS supports a batched write interface to amortize the I/O cost associated with each write over a larger number of pages. This allows applications to write many logical pages (LPAGE) using a single I/O, each LPAGE being uniquely identified via a logical page ID (LPID). The system described in this paper further, and unlike traditional SSDs where the page size is fixed, allows LPAGEs to have *variable sizes*. LPAGEs are aligned with 64 bytes to reduce the overhead for storing the LPAGE length. Thus, the smallest LPAGE size is also 64 bytes.

1) Write Buffer Semantics: Conventional SSDs with a BAAT interface only provide operational semantics, e.g., atomicity and durability, at the LPAGE level. Given that reading and writing a batch of LPAGEs differs from a block interface, the semantics provided by ELEOS is crucial. ELEOS guarantees atomicity and durability of write buffers. For atomicity, ELEOS ensures that either all LPAGES in a write buffer are persisted or none of them are. Durability ensures that committed write buffers are durable even after system crashes. ELEOS further ensures that pages within write buffers are posted to the SSD state in a serial order matching the order in which an application posted them to the write buffer. This is important because newer LPAGEs (submitted to the buffer later) are required to overwrite older ones for an application to see the same effect regardless of buffer size. However, ELEOS does not protect against subsequent data corruption in the storage media. Data corruption must be handled by applications, e.g., via replication.

Applications can read LPAGEs by simply specifying LPIDs. If a read is submitted after a write buffer has been acknowledged, ELEOS guarantees the read sees a state that is a prefix of a time ordered history that includes all the LPAGES of the write buffer. However, if a read is submitted concurrently with a write buffer, ELEOS guarantees only that the read sees a state that is a prefix of a time ordered history that includes a prefix

of the LPAGES in the write buffer, not a history that includes all pages in the entire buffer. We introduce a *system action* to describe our unit of atomicity. A system action ensures that the updates to logical pages in a buffer are either stored completely and become visible in the mapping table in the same order in which they occur within a buffer or are never visible in the mapping table. Applications have to perform additional concurrency control if stronger guarantees are desired.

2) Ordering Among Multiple Write Buffers: We define the order in which updates to pages occur when multiple updates for the same page are in a single write buffer. We need also to order updates sent in different write buffers, using different write I/O commands. Currently, SSDs do not provide an order guarantee if a second write is issued prior to a first write being acknowledged. Order can be enforced by waiting for a write to be acknowledged prior to issuing the next write. But, SSDs support parallel updates to flash storage and the performance benefit that it produces. Waiting for an ACK wastes parallelism and reduces write throughput/bandwidth. We want to avoid this limitation.

Our write protocol requires a user to open and close a session with the SSD, using a session ID (SID). SIDs are random numbers assigned by the SSD. Within a session, a user gives each write buffer a write sequence number (WSN). Each write requires an $< SSID, WSN >$ pair. WSNs start at 1 with each subsequent WSN one larger than the prior WSN. ELEOS applies the updates to SSD pages in WSN order within a session, and will not apply updates of a write until the updates of all prior writes with lower WSNs have been applied. The SSD will ACK writes in this same order. Users without ordering requirements can ignore sessions.

User sessions must survive controller crashes, requiring that the state of open sessions be durable. If a host system user, perhaps due to a controller crash, does not receive an ACK for a write, it can redo unACK'd writes. A write received at the controller having a WSN that is not one higher than the controller's remembered highest WSN is not applied, and the highest WSN is ACK'd to the host system. See section VIII for more about durability.

B. FTL Components

ELEOS implements a customized FTL to support batching and variable size pages. It uses a mapping table to store the latest physical address of each LPID and an EBLOCK summary table to manage the states of all EBLOCKs. All modifications to these two tables are made durable via logging and checkpointing. When a channel's storage utilization exceeds a "fill" threshold, garbage collection is triggered to reclaim storage space from a set of candidate EBLOCKs. Below we discuss the two system tables, i.e., the EBLOCK summary table and mapping table, in detail.

Each EBLOCK has an associated descriptor in the EBLOCK summary table. An EBLOCK descriptor contains the EBLOCK state (such as *FREE* if the EBLOCK is empty, *USED* if full, and *OPEN* if partially written), the erase count, the number of WBLOCKs containing data and the number of WBLOCKs containing metadata describing the data, the amount of available space (AVAIL), and a timestamp (TS).

The total size of an EBLOCK's descriptor is less than 32 bytes. For an 1TB SSD with EBLOCKs of size 8MB, for example, the EBLOCK summary table is smaller than 4MB, which can be easily cached in memory. However, the EBLOCK summary table is too large to be conveniently stored in the checkpoint record. To address this problem, the EBLOCK summary table is paginated and a small table that stores the latest location of each EBLOCK summary table page is introduced. This small table is less than 1KB, and can easily be stored in the checkpoint record.

The mapping table contains the latest physical flash address where the current LPAGE state (i.e., data) of an LPID is stored and the length of that data. Each physical address uses 8 bytes and stores the channel id, EBLOCK id, WBLOCK id, RBLOCK id, start offset and length of an LPAGE. We assume each LPAGE is stored continuously within a single EBLOCK, and the entire LPAGE can be retrieved by reading from the physical address stored in the mapping table.

The mapping table is usually too large to be totally cached in memory. For example, for an 1TB SSD with an average LPAGE size of 4KB, the entire mapping table occupies 2GB. To address this problem, we introduce two additional mapping table levels, namely a small table and a tiny table, on top of the mapping table and forming an index to it. The small table stores the physical addresses of mapping table pages and usually is a few megabytes. The small table is small enough to be cached, but it is still too large to be stored in the checkpoint record. Thus, an additional tiny table, which indexes the pages of small table pages, is introduced and it is completely stored in the checkpoint record.

IV. Write Path

We describe here how writes are processed in ELEOS. After a write buffer is received, ELEOS processes the pages of the write buffer completely with a system action. The system action has three phases, i.e., initialization, execution, and commit. During the initialization phase, a system action for the write buffer decides where the pages of the buffer are to be stored (called provisioning). These physical addresses on the storage media are used to generate I/O commands. Log records for the for the pages in the write buffer are generated to ensure that, should the controller crash, the controller state is either atomically updated or reset to the state prior to the write. During the execution phase, the I/O commands for LPAGEs and their log records are executed in parallel, transferring their data to the storage media. When execution is completed, the system action enters the commit phase by first forcing a commit log record, i.e., it issues the I/O for the commit log record and waits for the OCSSD to acknowledge that the commit record is durable. After the commit log record is durable, the system action is committed and the new physical addresses for the LPAGES are installed into the mapping table.

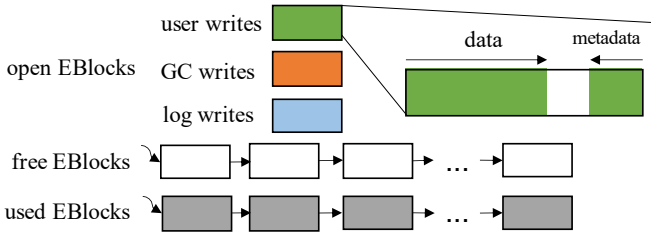

Fig. 3: Channel provisioning example: EBLOCKs are marked as open if they are partially written, free if they are empty, and used if they are full.

A. Initialization Phase

During the initialization phase, a system action is generated for the write buffer with three major tasks, i.e., write provisioning, generating I/O commands, and producing log records.

1) Write Provisioning: Write provisioning allocates physical addresses for LPAGEs that are in a write buffer. Since the storage media contains multiple channels, it is desirable to distribute user writes across all channels as evenly as possible to maximize I/O parallelism. Write provisioning is hence performed at two tiers, global provisioning and channel provisioning. Global provisioning partitions the write buffer into multiple chunks of approximately equal sizes so that each channel can receive roughly the same amount of data to write. Partitioning must respect the LPAGE boundaries to ensure that each LPAGE is written to contiguous storage in a single channel. After partitioning, channel provisioning allocates physical addresses to LPAGEs.

As shown in Fig. 3, each channel maintains the lists of used and free EBLOCKs. We maintain one open EBLOCK for each type of write, (1) log write, (2) new LPAGE write, or (3) garbage collection write. Log writes are separated from user writes because they use different GC mechanisms. GC writes are also separated from user writes so as to separate cold data moved by GC from hot data in the new user writes.

An EBLOCK stores both data, e.g., LPAGEs, and metadata. The metadata of an EBLOCK contains the type and LPID for each LPAGE, and is mainly used for GC (Section VI). When an open EBLOCK becomes full, its metadata is flushed to the last pages of the EBLOCK. Log records are written using this same metadata to enable the mapping table to be correctly updated should the system crash. After the metadata is successfully persisted, the EBLOCK is then closed and enters the used state. When an EBLOCK for user writes is closed, its timestamp is set as the current time, which is the update sequence number[1] of the last page being written.

Write provisioning is performed at the WBLOCK granularity. To provision physical addresses for a batch of LPAGEs, we increment the current data WBLOCK position in the EBLOCK and populate the physical address of each LPAGE. The LPIDs and types are added to the EBLOCK metadata. During the provisioning process, an EBLOCK may not have sufficient storage to fully contain the LPAGES. In this case, the EBLOCK is closed by flushing its metadata to the last unused pages of the EBLOCK. The EBLOCK only flushes its metadata after all data WBLOCKs have been persisted so that the metadata occurs in the highest order pages of the EBLOCK and describes all data pages of the EBLOCK. During recovery, this allows us to safely conclude an EBLOCK has been successfully closed after checking that its metadata has been persisted. During garbage collection, only the metadata pages need to be read to decide which data pages remain valid and hence moved elsewhere.

Thus, instead of closing an EBLOCK immediately during provisioning, we let the system action that causes an EBLOCK to be full submit an I/O command to flush the EBLOCK's metadata after submitting all I/O commands to write data WBLOCKs. After an open EBLOCK is full, a new EBLOCK is taken from the free EBLOCK list. When the number of free EBLOCKs is low, e.g., lower than 10%, the channel will be marked for GC to reclaim space from used EBLOCKs.

Since provisioning is done at the WBLOCK level, and LPAGEs are variable length, there may be some storage fragmentation in WBLOCKs. First, for a chunk of LPAGEs allocated to each channel, the last WBLOCK can be fragmented if the LPAGEs cannot completely fill it. Second, if an EBLOCK does not have enough space to store the next LPAGE during provisioning, the remaining space will be fragmented. The storage space lost by fragmentation will be added to AVAIL of the EBLOCK in the EBLOCK summary table. Given that an EBLOCK is very large (MBs), the fraction of space lost to fragmentation is usually very small.

2) I/O Command Generation: After provisioning physical addresses for LPAGEs, we then generate I/O commands to write in-memory LPAGEs to the OCSSD. Each I/O command writes a contiguous chunk of memory to a single WBLOCK. With provisioned physical addresses, the generation of I/O commands is straightforward: we can generate I/O commands for each provisioned WBLOCK based on the offset of the first LPAGE in this EBLOCK.

Consider the example write buffer in Fig. 4, where each WBLOCK is 16KB. Suppose LPAGEs 1 to 3 are provisioned to the first 3 WBLOCKs in EBLOCK 1, and LPAGEs 4 to 6 are provisioned to the first 2 WBLOCKs in EBLOCK 2. Then five I/O commands are generated to write these LPAGEs, one for each WBLOCK. For example, the I/O command for WBLOCK 1 of EBLOCK 1 copies the first 16KB of the write buffer, while the I/O command for WBLOCK 1 of the EBLOCK 2 copies 37KB, based on the offset of LPAGE 4, to 53KB of the write buffer. Note that the last provisioned WBLOCK at each EBLOCK may copy some additional memory that is not provisioned to that EBLOCK, e.g., WBLOCK 3 of EBLOCK 1. This is simply treated as internal fragmentation of these WBLOCKs with the fragmentation size added to AVAIL for the EBLOCK.

3) Logging: Along with generating I/O commands, log records are produced for the write buffer as well to ensure the durability of changes produced by committed system actions. The log records only contain the redo information

[1]We use update sequence numbers as our proxy for time.

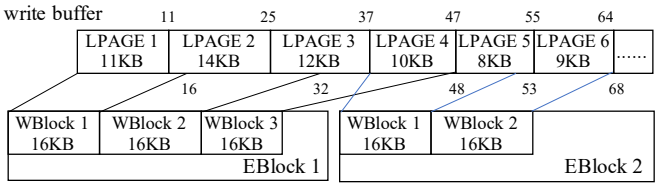

Fig. 4: I/O command generation example

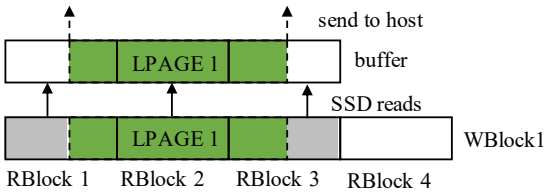

Fig. 5: An example of reading a variable size LPAGE

of the changes made to the mapping table and EBLOCK summary table. No undo information needs to be logged because ELEOS follows a no-steal policy [16]. Specifically, each LPAGE written results in a log record to store its LPID and new physical address. We will further discuss how the changes to the mapping table and EBLOCK summary table are redone during recovery in Section VIII.

B. Execution Phase

After the system action is created for a write buffer, the I/O commands are executed at all channels in parallel. Specifically, we use one submission queue for each channel to execute I/O commands in order. Within each EBLOCK, I/O commands are always executed in submission order.

We use one byte per I/O command to keep track of the status of each I/O command. When the I/O command completes, either having succeeded or failed, the system action is notified to update its status. If any I/O command fails, the system action is aborted and the user must retry writing the buffer.

C. Commit Phase

A system action is aborted or committed based on whether all I/O commands are successfully completed. Aborting a system action is straightforward. We simply treat the provisioned physical addresses for this system action as garbage by incrementing AVAILs of provisioned EBLOCKs. The wasted space will be eventually reclaimed by garbage collection.

Recall that we ensure system actions are committed within a session in WSN order. When a system action is about to commit, i.e., the system actions associated with all lower WSNs have committed, a commit log record is forced that contains the current WSN. After the commit log record is durable, the system action can then install the new physical addresses for LPAGEs into the mapping table. The space occupied by the over-written versions of LPAGES is added to AVAIL of the containing EBLOCKs in the EBLOCK summary table. The space occupied by these obsolete LPAGES will eventually be reclaimed by GC.

V. Read Path

To read an LPAGE, the user specifies an LPID. After receiving the read request, ELEOS accesses the mapping table to get the physical address of the LPAGE. Recall that the physical address contains the start address and the length of an LPAGE. Based on the physical address, ELEOS generates multiple I/O commands to transfer the LPAGE from the storage media to memory. Since SSD reads are performed in the RBLOCK level, which, in general, is not aligned with LPAGE, some extra data may be transferred to memory as well. Finally, based on the start offset of the LPAGE in the first RBLOCK and its length, ELEOS transfers the exact content of the LPAGE back to the host.

Consider the example in Fig. 5. Suppose LPAGE 1 is stored in RBLOCKs 1 through 3. To read this LPAGE, these RBLOCKs are first transfered to memory. Then based on the start offset and length stored in the physical address of LPAGE1 in the mapping table, only LPAGE1 is transferred back to the host. This avoids transferring extra data and also ensures that data from physically adjacent LPAGEs is not revealed, avoiding a possible security compromise.

VI. Garbage Collection

Log structured stores require garbage collection to reclaim storage space occupied by obsolete pages. In ELEOS, GC is performed on a channel when the number of free EBLOCKs is below a defined threshold. GC erases partially full EBLOCKs so that these EBLOCK can be reused again. Before erasing such an EBLOCK, we must move all still current LPAGEs stored in this EBLOCK to new locations. Thus, it is critical to select EBLOCKs to minimize this data movement cost. This section discusses how GC is implemented in ELEOS.

A. EBLOCK Selection

Wisely selecting EBLOCKs to GC is critical to reduce the overall GC cost, i.e., to minimize the amount of data moved per EBLOCK erased. Some simple selection strategies include selecting oldest EBLOCKs or selecting EBLOCKs with most available space (largest AVAIL). LLAMA [10] uses the first strategy by organizing the storage as a circular buffer. However, this strategy is only optimal for uniformly distributed page updates since the oldest EBLOCK has, on average, received the most updates. The second strategy only locally optimizes the immediate GC step, but often fails to minimize the overall GC cost. Consider two EBLOCKs, E1 and E2, with similar available space. E1 contains hot data and is updated more frequently than E2. To minimize the overall GC cost, we should select E2 to GC first because its available space would not increase as much as the available space in E1.

To minimize the GC cost, ELEOS implements the minimum-cost-decline strategy [5]. Its key idea is select EBLOCKs that have the smallest expected decline in GC costs. That is, we would expect the least gain by further delaying GC of these EBLOCKs. To select EBLOCKs to GC, we compute a score for each EBLOCK as $\frac{1-E}{E^2 \times age}$, where E is the fraction of the available space in an EBLOCK, and age is defined as

the elapsed time since the EBLOCK's timestamp. EBLOCKs with smallest computed scores are selected to GC. Noted that EBLOCKs for storing log records are GCed separately as these EBLOCKs are erased via log truncation. EBLOCKs that store truncated log records will always have "smallest scores" because no data movement is needed.

B. Separating Cold from Hot

In practice, user writes are usually a mix of hot data with cold data together. Since GC is performed at EBLOCK level, it is important to separate hot data from cold data so that data movement is minimized. We use a simple separation strategy of grouping LPAGEs by their ages. User writes are assumed to be hotter (these are new updates) and are stored into EBLOCKs dedicated to user writes. GC writes, i.e., for LPAGEs that are not updated recently, are stored into separate EBLOCKS because they represent colder data.

To improve the effectiveness of cold-hot separation, we further separate GC writes so that LPAGEs with similar update frequencies are stored together. The basic idea is to use multiple opened EBLOCKs. Each opened EBLOCK uses its timestamp to approximate the creation timestamps of stored LPAGEs. Thus, for each EBLOCK to be GC'd, we write all of its valid LPAGEs into an opened EBLOCK with the closest timestamp. This allows us to separate EBLOCKs based on their age, which approximates their update frequency.

C. Moving Valid LPAGEs

Moving valid LPAGEs involves reading them into memory and writing them to new locations. We reuse most of the codepath for handling user writes (Section IV) to process GC writes. A system action is formed for each EBLOCK being GC'd. Once the system action is successfully committed, all GC writes of valid pages in an old EBLOCK are durable and can be erased. In what follows, we focus on some differences in the handling of GC writes.

To GC an EBLOCK, we first determine which LPAGEs are still valid. The EBLOCK's metadata, i.e., in its last few WBLOCKs, stores the LPID and type for each stored LPAGE. For each LPAGE type, we check the latest physical address in the mapping table via its LPID. An LPAGE is valid if it is still stored in this EBLOCK, and hence must be read into memory to be moved.

An EBLOCK may store multiple versions of an LPAGE with multiple instances of its LPID in the metadata. Even though these LPIDs will always point to the same physical address, because the LPID will find the same value in the mapping table, we should only move this LPAGE once. Hence we exploit an important property of write provisioning, i.e., that for any two valid LPAGEs P1 and P2 in an EBLOCK, if P2 is newer than P1, then P2's address must be after P1's address. The update order of LPAGEs is thus determined by their positions in the EBLOCK's metadata, where more recent LPAGEs have larger positions. Thus, if we process LPAGEs from newest (largest physical address) to oldest, the physical addresses of valid LPAGEs must be monotonically decreasing.

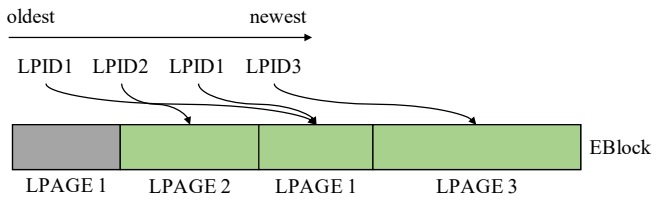

Fig. 6: GC EBLOCK example

Otherwise, the metadata entry for an LPAGE must be invalid, even though its mapping table physical address may still point to this EBLOCK.

Consider the example in Fig. 6. The EBLOCK stores four LPIDs. Each LPID points to the physical location where it is stored in the EBLOCK, located via examination of the mapping table entry. LPIDs are processed from newest to oldest, namely LPID3, LPID1, LPID2, LPID1. Based on the discussion above, the last LPID1 is invalid because its physical address is found to be larger than its immediate predecessor, and can be safely ignored.

A system action is formed to write all valid LPAGEs into new locations, as we did for user writes (Section IV). Committing a GC system action modifies the mapping table to install the new locations of moved LPAGEs. This is implemented as a set of conditional updates, i.e., a new location is only installed if the same LPAGE has not been updated by user writes that occurred between the time GC examined LPAGEs for validity and the time the GC system action changes are committed and installed. Otherwise, the LPAGE relocation is aborted because GC has moved a now invalid version of the LPAGE.

VII. Handling Write Failures

Writing a WBLOCK may fail. This may be due to limited SSD writes or simply variations in SSD fabrication. This is an example of a problem that host-based log structuring implementations do not have to deal with, as they "leave it to the SSD". But our controller is part of the SSD, and as such, it must deal with write failures.

When a write failure occurs, we abort the system action (either GC or user) so that the caller can retry. However, when a WBLOCK cannot be written, subsequent WBLOCKs of the same EBLOCK cannot be written either. Thus, the EBLOCK metadata can no longer be written to the storage media, making the entire EBLOCK unusable, including LPAGEs written to the EBLOCK prior to the write failure. To address this problem, when a write fails, we must migrate the previous committed LPAGEs in this EBLOCK to new locations.

The implementation of EBLOCK migration is very similar to GC, which allows us to reuse most of the codepath of GC to handle write failures. There are two main differences that require care. First, when an EBLOCK is migrated, all subsequent I/O commands writing to this EBLOCK must be failed as well. Second, the EBLOCK to be migrated may have some LPAGEs written by uncommitted system actions. If the EBLOCK is migrated before those system actions commit, the system actions may install the old addresses pointing to

the migrated EBLOCK upon commit. To address this, before an EBLOCK is migrated, those uncommitted system actions are aborted as well. Since write failures are very rare and each EBLOCK usually contains a very small number of active system actions (due to large write buffers), aborting system actions due to write failures would have at most modest impact. Because this approach is simple, there is almost no impact on normal case overall performance of ELEOS.

VIII. Durability

The committed system actions and user sessions must be durable and continue across controller crashes. We use logging and checkpointing techniques to ensure the durability of three tables, i.e., the mapping table, the EBLOCK summary table, and the session table, together with the metadata of opened EBLOCKs because it is not yet durable in the EBLOCK. LPAGEs do not need to be logged because a system action is only committed after all LPAGE writes contained in it are successfully completed. If a system action is not committed before the system crashes, it is aborted during recovery. In the remainder of this section, we describe the logging, checkpointing, and recovery techniques used in ELEOS.

A. Logging

The log in ELEOS is implemented as a linked list, where each log page, i.e., a WBLOCK, stores a pointer to the successor log page. Writing a log page may fail. When this happens, the successor log page is written to a different location, which breaks the log chain because the previous log page now points to an invalid location. To overcome this difficulty, we provision the next three locations for the successor log page. Pointers to these pages are stored as forward pointers in the predecessor log page. During recovery, we read from these three locations one by one until the first valid log page is found. When a log page cannot be written to any of these three locations, we currently shut down writing to the SSD.

All writes in ELEOS, including user writes, GC writes, and checkpointing writes, are handled using system actions. They follow the same logging protocol by first producing log records of LPIDs with new addresses before actually writing WBLOCKs for both data LPAGEs and log LPAGEs. Once the WBLOCKs for these LPAGEs are successfully durable, a commit record is forced. Hence, ELEOS uses the same protocol to recover all writes.

B. Checkpointing

ELEOS regularly performs fuzzy checkpointing to bound the recovery time and truncate log records. In ELEOS, the log truncation LSN is determined by the minimum of three factors, (1) smallest LSN of active system actions, (2) smallest LSN of the mapping table and the EBLOCK summary table, and (3) smallest LSN of all open EBLOCKs. To truncate log records, checkpointing flushes dirty pages of the mapping table and the EBLOCK summary table and forcibly closes some open EBLOCKs if they are opened for too long. Forcibly closing open EBLOCKs is necessary because EBLOCKs for storing GC writes may stay open for a very long time, as ELEOS maintains multiple open EBLOCKs for GC to separate hot data from cold data.

The checkpointing process is as follows. First, the log truncation LSN is determined. Next, the dirty pages of the mapping table and the EBLOCK summary table are flushed. The session table is flushed in its entirety. Checkpointing flushes one dirty WBLOCK at a time, enabling each WBLOCK to be completely full. Each such WBLOCK write is logged so that the location of these parts of the table can be found during recovery. After all writes are completed, a new checkpoint record is created and persisted to a "well-known" location. Checkpointing does not itself truncate the log. Rather it only updates the log truncation LSN. The EBLOCKs that store earlier log records will be erased later by GC.

C. Recovery

During recovery, we restore the system tables and the metadata of open EBLOCKs. The recovery log contains the log records for user writes, GC writes, and checkpointing writes. When an EBLOCK is closed, we write a log record describing this to reduce the number of potentially open EBLOCKs. If an EBLOCK is closed but without a "close" log record, we examine it as if it were open.

The basic idea of recovery is the same as in database recovery – first read the checkpoint to initialize the system state, then replay the log records to bring the system up-to-date. When replaying log records, we only need to redo updates without any undo because only committed system actions modify the state. Moreover, the EBLOCK states changed by aborted system actions, e.g., provisioned WBLOCKs, are not undone. Whatever such system actions wrote will be counted in AVAIL as garbage and subject to GC.

Even though the basic idea of recovery is straightforward, we encountered a number of technical issues, which are discussed below.

1) Two-Pass Log Replay: Before redoing updates from the log records, we need to locate the mapping table and the EBLOCK summary table from the stored checkpoint information. The basic problem we face is that (1) the checkpointed state is mixed in with the recovery log and (2) the location of the checkpoint state that can be reached from the checkpoint record we write at the end of a checkpoint may point to physical addresses that have been moved by garbage collection. GC is used to reclaim space in all EBLOCKs, including those with checkpoint information. The moved checkpoint information will not be reachable via the physical addresses that we saved in the checkpoint record.

We illustrate this problem in Fig. 7. Initially, suppose a mapping table page Page1 is stored in physical address Addr0. A user system action UserAct1 executes that modifies Page1, with the change tracked by a log record describing this. Later, a GC system action GCAct1 executes and moves Page1 to a new physical address Addr1. Finally, the system crashes after these two system actions commit. During recovery, by checking the

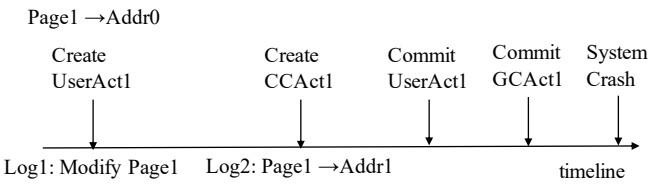

Fig. 7: Example of moved system table pages by GC

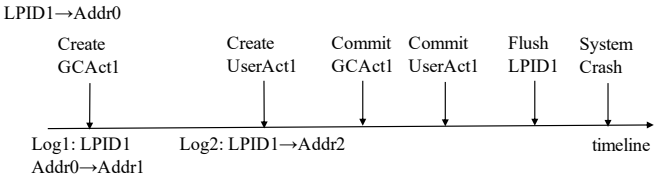

Fig. 8: Example system actions causing inaccurate AVAILs

checkpoint record, we assume Page1 is still stored in the old address Addr0, which has already been erased.

To deal with this, we institute a two-pass replay of the log. The first pass recovers only the physical addresses of mapping table and EBLOCK summary table pages that are stored as part of the checkpointed state. The second pass over the log applies updates to the values stored in these tables as a result of system execution since the start of the checkpoint (i.e., from the log truncation LSN).

2) Mapping Table and AVAIL: To Recover the mapping table, we redo all committed system actions. During normal operation, updates made by user system actions are always installed into the mapping table, while updates made by GC system actions are installed only if the old address in the mapping table matches the old address stored in the log record. We use this same logic during recovery. Redo recovery of the mapping table is idempotent because we have logged the after-state of the operation. This guarantees we will always get the latest version of the mapping table when recovery completes.

When committing a system action, we need to ensure that the storage consumed by the old versions of updated LPIDs is garbage collected. We do this by incrementing AVAIL of old EBLOCKs, eventually leading to their garbage collection. We need to do this as well during recovery, but the same mechanism does not work during recovery because the mapping table may not contain the correct old address at the time that recovery redoes an operation.

Consider the example in Fig. 8. LPID1 initially points to Addr0 in the mapping table. Then, two user system actions, namely UserAct1 and UserAct2, modify LPID1 with new addresses Addr1 and Addr2 respectively. After their commit, the mapping table page containing LPID is flushed, e.g., by page eviction or checkpointing, followed by a system crash. At recovery, redoing the update of UserAct1 will not identify the correct prior old address, believing it to be Addr2, instead of the correct old address, Addr0. The same problem arises for GC system actions.

An inaccurate EBLOCK AVAIL may significantly impact the GC efficiency after recovery since we choose EBLOCKs to erase based to a substantial degree on the available space that can be reclaimed. To maintain AVAIL across system crashes, we need to log additional information. However, as mentioned before, logging old addresses during the initialization phase is difficult because uncommitted system actions may not have applied their changes yet.

To minimize the error in AVAIL, each user and checkpoint system action produces additional log records that contain old addresses for LPIDs and lazily writes them. For GC system actions, only aborted LPIDs are logged because old addresses have already been logged. No additional mapping table lookups are required because old addresses can be obtained directly when installing new addresses. After all log records are produced, a DONE record is logged signaling no more log record will be produced for the system action.

During recovery, we use these additional log records, whether accompanied by a DONE or not, to update EBLOCK AVAIL. Should the system may crash before a DONE record is written, some EBLOCK AVAILs may not be 100% accurate. However, this reduces the potential error in AVAILs, which is important as EBLOCK GC ordering is impacted by the value of its AVAIL.

3) EBLOCK Summary Table & Metadata: Recall that each EBLOCK stores its state, currently provisioned WBLOCKs, available space AVAIL, and timestamp in the EBLOCK summary table. Each open EBLOCK also has metadata with LPIDs and types for stored LPAGEs. Unlike the mapping table, redoing updates on the EBLOCK summary table is not idempotent because we do not log the entire value of an EBLOCK state in each log record. To enable idempotent recovery, we store the flush LSN for each EBLOCK summary table page. A case analysis below describes how this works.

Case 1: A log record REC_W writes to some physical address in EBLOCK E. If E's state is not open and the flush LSN is no less than REC_W's LSN, then REC_W can be safely ignored. Otherwise, we first add the LPID stored in REC_W into E's metadata. We do not check the LSN because E's LSN only protects the EBLOCK summary table, not the metadata. If REC_W's LSN is greater than E's flush LSN, we further update the EBLOCK summary table by redoing the provisioning for the new address stored in REC_W. Recall that we use a single thread to execute the initialization phase of system actions. Thus, the addresses stored in the log records will be in the same order as they are provisioned.

Case 2: A log record REC_C closes an EBLOCK E. Again, REC_C is ignored if E is already closed and REC_C's LSN is smaller than E's flush LSN. Otherwise, we clear E's in-memory metadata and set E's state to closed in the EBLOCK summary table.

Case 3: A commit log record for a system action is encountered. When a system action is committed, the storage associated with the old addresses is added to AVAIL of the relevant EBLOCK, as discussed in Section VIII-C2. For a system action that is aborted, either because an abort log record is encountered or because no commit log record is encountered, it is the storage associated with the new addresses

that is added to AVAIL instead. Note that this process is protected by LSNs, i.e., this is only redone if the log record's LSN is larger than an EBLOCK's flush LSN.

Unfortunately, even after all log records are replayed, we cannot guarantee the EBLOCK summary table is up-to-date because the system may crash before log records for a system action are persisted but some WBLOCKs have already been written. In this case, writing to non-empty WBLOCKs would cause a write failure. We alleviate this problem in two ways without sacrificing write performance. First, for all open EBLOCKs after recovery, we fix the current WBLOCK position by reading forward until we encounter the first empty WBLOCK. The storage size of the non-empty WBLOCKs is added to AVAIL as if they were written by aborted system actions. However, it is still possible that some EBLOCK is opened by aborted system actions but no log record is persisted. Performing a full scan over all EBLOCKs is too expensive. We expect these EBLOCKs are very rare because each channel only maintains a few open EBLOCKS at all times. Moreover, writing log records is much faster than writing LPAGEs because each system action only produces a small amount of log records compared to its LPAGEs. Thus, we simply choose to tolerate write failures caused by writing to non-empty WBLOCKs as described in Section VII.

IX. Evaluation

In the evaluation we are primarily interested in comparing the impact of our new batch I/O storage interface with the existing block I/O interface, and showing the benefit of supporting variable size pages compared to fixed size pages. We thus focus our experimental study on a single-threaded experiments in order to isolate the performance impact of the storage interface.

A. Configuration

1) Hardware: We performed our evaluation on an Ubuntu 16.04.6 LTS host machine with an Intel Xeon E5-1620 3.5GHz processor with 32GB of DRAM. Our programmable SSD is built based on a standalone platform, named STT100, manufactured by Broadcom for developing storage applications. The platform uses BCM5880X SoC equipped with an ARM Cortex-A72 1.8GHz processor and runs Ubuntu 16.04.6 LTS as a working OS. For issuing read, write, and erase operations against raw flash memory, we used a CNEX Open-Channel SSD attached to the STT100 via PCIe Gen3x8. Data transfer between the host and the programmable SSD is performed via stream sockets with the NVMe-oF/TCP protocol. We configured the network speed between the host and the programmable SSD to be 100Gbps, which guarantees that the network is not a performance bottleneck in all experiments.

2) Software: On the programmable SSD, we ran OX [1], a software framework for programming a storage controller. The framework provides a full-fledged, generic FTL to enable reads and writes via a standard block-based interface. On top of the framework, we implemented ELEOS, a customized FTL to support batched writes of variable size pages of an arbitrary number of bytes. This new implementation relies on in-SSD log structuring with the following new APIs: (1) $\text{read}_{\text{LPID}}$ for reading a variable size page with its LPID, which differs from the standard block read (used in OX-Block) that requires a starting address of the page, (2) $\text{flush}_{\text{batch}}$ for flushing a batch of variable size pages entirely. Compared with the standard write of an array of bytes, ELEOS identifies the pages by parsing the batch using metadata within the batch.

3) Benchmarks: We used the following two sets of benchmarks. First, we ran an I/O trace collected from running the TPC-C benchmark with the scale factor 1000 on the B^+-tree storage engine of Apache AsterixDB [17]. The page size was set at 4KB. We enabled page compression of the B^+-tree so that the produced I/O trace contains variable size pages. The average size of compressed pages was 1.91KB. The I/O trace was collected during the running phase of the TPC-C benchmark, after the base tables are loaded. Finally, we used the first 100GB pf page writes in our evaluation.

Second, we ran a key-value store based on Bw-tree [9] with a set of YCSB workloads [2] that is widely used for evaluating performance of NoSQL stores. It should be noted that the original Bw-tree avoids updating base pages directly by maintaining a delta chain that stores modifications to a node. When the chain becomes too long, a compaction operation is needed to consolidate delta records into the base page. Therefore, we modified the original Bw-tree to simply perform updates in-place without creating delta chains. In addition, with ELEOS the Bw-tree no longer need (1) remember where LPAGEs are located on the SSD (i.e., checkpointing and recovering its mapping table are not needed because cached LPAGES are only mapped to their main memory locations) and (2) perform garbage collection that is now done by the SSD controller, which is aware of LPAGE physical locations..

Our YCSB workload is write-heavy, containing 5% reads and 95% updates[2]. ELEOS batching currently works only on the write path to the SSD. Keys for read and update operations are selected randomly from the set of existing keys in the index according to a Zipfian distribution. Our YCSB dataset has 10 million unique records, each consisting of an 8-byte key, and a 100-byte payload. For each run of benchmarks we fully re-initialized an index with records in the dataset, and then ran the specified workload for 300 seconds. We report the total number of operations completed in that time, where operations are either reads or updates. The read and update operations are interleaved. Specifically, we performed 19 updates, then 1 read, then repeated the cycle. We varied the buffer cache size of the Bw-tree over a range measured as a percentage of the dataset size to evaluate its throughput while shifting the workload from mostly in-memory to mostly on SSD. The maximum Bw-tree page size was set to 4KB, and the write buffer size used in the Bw-tree flush operation was set to 1MB.

[2]We also evaluated a read-heavy workload with 95% reads and 5% updates, but omit the results due to space constraints.

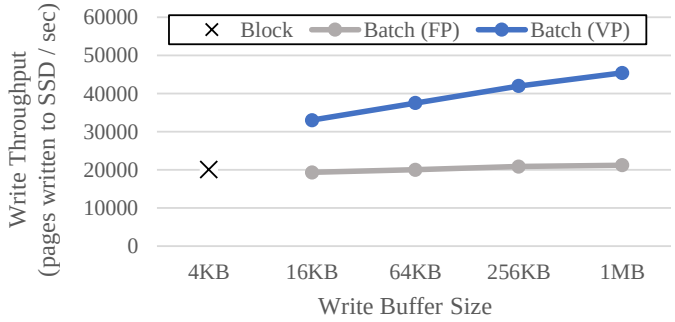

Fig. 9: TPC-C write throughput, varying the write buffer size for the batch interface. FP and VP denote fixed- and variable size pages, respectively.

	Block	Batch (FP)	Batch (VP)
Write Throughput (pages/sec)	52.73K	255.03K	447.79K
Write Bandwidth (MB/sec)	206.17	1015.86	992.39

TABLE II: TPC-C write throughput with a programmable SSD simulator using a high-end CPU. Note that the write buffer size for the batch interface was set to 1MB.

B. B^+-tree with TPC-C

To evaluate the impact of variable size page support over a batch interface, we carried out experiments of replaying the TPC-C trace to measure the write throughput (i.e., the number of TPC-C pages written to the SSD per second). The results with different batch sizes are shown in Fig. 9. As one can see, it is clear that batching writes is effective (over a block-oriented approach), and its impact is greater as the larger batch size is used. It was also observed that by removing internal page fragmentation, the batched write interface of variable size pages provided approximately twice the write throughput of the batch interface with fixed-size pages.

In this experiment, interestingly, the programmable SSD easily became a performance bottleneck, mainly due to the relatively weak storage controller[3]. For example, as mentioned in Section IX-A3 a commercial SSD's FTL runs in firmware on a custom ASIC while our FTL of the programmable SSD runs on a storage controller running Linux as well. In addition, the programmable SSD uses a standard network protocol (i.e., TCP/IP) for NVMe over fabric as a flexible solution for the communication protocol. Unfortunately, the socket connection delivers fast network performance at the cost of moving data on the network stack with a high CPU utilization (e.g., more than 60% of CPU loads were used for the socket communication for some experiments). Such high utilization might not be ideal for storage devices where relatively fewer and weaker cores exist compared to high-performance servers[4].

[3]In all cases, the I/O bandwidth was about 85MB/sec, and the host CPU utilization was less than 15%.

[4]Exploring other advanced protocols such as Remote Direct Memory Access (RDMA) to reduce the network overhead is a topic for future work.

To show the full potential of a batch interface (avoiding being I/O bound), the same experiment was carried out using a remote server where a programmable SSD simulator runs with a high-end CPU. As can be seen in Table II, the batch interface showed about 8.5 times higher write throughput than the block interface when the performance bottleneck moved from storage to CPU. In addition, given that the average size of the TPC-C variable size pages was slightly less than 2KB, about half of the write bandwidth of the batch interface supporting only fixed-size pages was wasted writing unused space within the page.

C. Bw-tree with YCSB

1) Normal State: In this experiment, we evaluated the overall throughput of the Bw-tree in a non-durable setup where both checkpointing and garbage collection are disabled. The results with different cache sizes are shown in Fig. 10(a). In all cases, as expected, the overall throughput decreases when the buffer cache size becomes smaller because of more cache misses, resulting in the increased number of I/Os issued to the SSD. We can see clearly that the variable size implementation does not degrade the performance compared to the one achieved when only dealing with fixed-size pages. This is notable as flash page alignment is lost when supporting variable size pages. At the same time, the variable size page implementation reduces by about 30% the total amount of data written by packing variable size pages without internal page storage fragmentation as shown in Fig. 10(b).

Under the same hardware configuration, we could observe even clearer benefits of batching writes over a block-at-a-time interface. As shown in Fig. 10(a), Batch outperformed Block by 1.12 – 1.97×, depending on the configured cache size. The main reason for this behavior is due to the different write granularities supported by each configuration. Unlike the read path of the Bw-tree where a single page is read at a time, in the write path a 1MB-sized write buffer is flushed to the SSD during the benchmarks. Once the flush starts, the 1MB data is first split into 17 packets[5] according to the NVMe-oF/TCP protocol, and then the packets are sent to the SSD.

On the SSD side, Batch waits until all 17 packets are received, and then creates a single write context to guarantee the atomicity of the whole 1MB data. In contrast, Block does not know any logical relationship among the 17 packets, so a write context needs to be created per each packet, resulting in 17 contexts for the 1MB data. This means that Block has to process about 17× more internal writes than ELEOS, resulting in much more commit log records that need to be generated and flushed before completing the 1MB write request. In addition, the small write granularity of OX prevents fully exploiting the internal SSD parallelism - the maximum size of an internal write of Block is bounded by the packet size, which is not big enough to leverage all flash channels at once.

[5]Note that the maximum size of an IP datagram, a basic transfer unit associated with a packet-switched network is 65,532 bytes including a 20 bytes header followed by a data area.

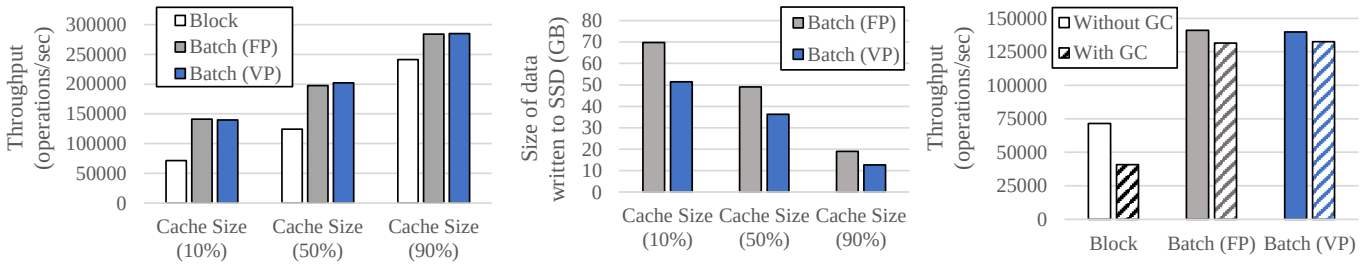

Fig. 10: (a) Bw-tree throughput with a 1MB write buffer, varying the cache size. (b) The total amount of data written to the SSD during benchmarks. (c) Bw-tree throughput when enabling garbage collection when the cache size is set to 10% of the dataset size.

2) Garbage Collection: We then explored how garbage collection (GC) affects the overall performance. To facilitate the GC process, the SSD capacity was limited to 10 times larger than the dataset size with an over-provisioning of 30% to avoid the situation where all channels are full. This over-provisioning space, designated as "free space", assists in efficient delivery of free blocks during GC, helping to increase the lifetime, endurance, and overall performance of the SSD. GC is triggered when the number of used blocks in the SSD reaches to a certain threshold. In this experiment, the GC process described in Section VI is conducted by the SSD controller when the SSD space is 90% full.

On the other hand, when running with a conventional block-oriented interface, the Bw-tree requires an additional host-based GC process that continuously reclaims space occupied by stale data to ensure contiguous free areas for appending buffers with new versions of pages. Since versions of pages have different lifetimes, very old parts of the log could contain still current pages. To reuse this old section of the log, the still current pages need to be moved to the active tail of the log, appending them there so that the old part can be recycled for subsequent use. In other words, the oldest part (head of the log) is "cleaned" and added as new space at the active tail of the log where new page state is written. Note that Batch does not require this host-based GC as mentioned in Section IX-A3

The results when the 10% cache size (small enough to aggressively require writes to the SSD) are shown in Fig. 10(c). As expected, compared with the case where GC is completely disabled we observed performance degradation because a fraction of CPU and I/O resources used to be dedicated for handling the benchmark operations needs to be used for performing GC periodically. However, as can be seen in the figure, Batch performed GC much more efficiently than when using the SSD with a block-oriented interface. For example, with variable size pages, the overall throughput of the Bw-tree declined by about 5.2% while performance of Block declined by about 42.3%. Batch exactly knows which flash-resident data is garbage, so during GC Batch needs only to move valid data. However, the Bw-tree with Block lacks such information, and therefore needs to read whole LS segments, and then parse them them to figure out which data is valid, resulting in a significant increase in the read amplification of its GC process.

X. Conclusion

In this paper, we have described the implementation of ELEOS, a customized SSD controller supporting batched writes of variable size pages. This work is an example of exploiting the new opportunities as to where functionality is implemented and how it can simplify systems in the new world of open hardware architectures.

ELEOS eliminates the log structuring overhead from the host to improve the performance of host applications. In particular, the host burden of recovery and garbage collection for a log structured store is entirely removed. Moreover, ELEOS's native support for variable size pages improves I/O performance by eliminating internal page fragmentation, hence reducing the amount of data written.

References

[1] OX: Computational storage SSD controller. https://github.com/DFC-OpenSource/ox-ctrl, 2019.

[2] B. F. Cooper et al. Benchmarking cloud serving systems with YCSB. In *SoCC*, pp. 143–154, 2010.

[3] B. Gu et al. Biscuit: A framework for near-data processing of big data workloads. In *ISCA*, pp. 153–165, 2016.

[4] D. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *DaMoN*, pp. 1–10, 2018.

[5] D. Lomet et al. Efficiently reclaiming space in a log structured store. arXiv:2005.00044, 2020.

[6] J. Do et al. Query processing on smart SSDs: opportunities and challenges. In *ACM SIGMOD*, pp. 1221–1230, 2013.

[7] J. Do et al. Programmable solid-state storage in future cloud datacenters. *CACM*, 62(6):54–62, 2019.

[8] J. Do et al. Improving CPU I/O performance via SSD controller FTL support for batched writes. In *DaMoN*, pp. 1–8, 2019.

[9] J. Levandoski et al. The Bw-tree: A B-tree for new hardware platforms. In *ICDE*, pp. 302–313, 2013.

[10] J. Levandoski et al. LLAMA: A cache/storage subsystem for modern hardware. *PVLDB*. 6(10):877–888, 2013.

[11] M. Rosenblum et al. The design and implementation of a log-structured file system. *ACM TOCS*, 10(1):26–52, 1992.

[12] M. Cornwell. Anatomy of a solid-state drive. *ACM Queue*, 10(10), 2012.

[13] M. Bjørling et al. Lightnvm: The linux open-channel SSD subsystem. In *FAST*, pp. 359–374, 2017.

[14] P. Wang et al. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. In *EuroSys*, pp. 16:1–16:14, 2014.

[15] S. Seshadri et al. Willow: A user-programmable SSD. In *OSDI*, pp. 67–80, 2014.

[16] T. Haerder et al. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

[17] S. Altwaijry et al. AsterixDB: A scalable, open source BDMS. *PVLDB*, 7(14):1905–1916, 2014.