Noname manuscript No.
(will be inserted by the editor)

Better Database Cost/Performance via Batched I/O on Programmable SSD

Jaeyoung Do · Ivan Luiz Picoli · Philippe Bonnet · David Lomet

the date of receipt and acceptance should be inserted later

Abstract Data should be placed at the most cost and performance-effective tier in the storage hierarchy. While performance and cost decrease with distance from the CPU, the cost/performance trade-off depends on how efficiently data can be moved across tiers. Log structuring improves this cost/performance by writing batches of pages from main memory to secondary storage using a conventional block-at-a-time I/O interface. However, log structuring incurs overhead in the form of recovery and garbage collection. With computational Solid-State Drives, it is now possible to design a storage interface that minimizes this overhead. In this paper, we offload log structuring from the host to the SSD. We define a new batch I/O storage interface and we design a Flash Translation Layer that takes care of log structuring on the SSD side. This removes the CPU computational and I/O load associated with recovery and garbage collection. We compare the performance of the Bw-tree key-value store with its LLAMA host-based log structuring to the same key-value software stack executing on a computational SSD equipped with a batch I/O interface. Our experimental results show the benefits of eliminating redundancies, minimizing interactions across storage layers, and avoiding the CPU cost of providing log structuring.

Keywords Key-value store, Log structuring, Programmable SSDs, Bw-tree, LLAMA

Jaeyoung Do · David Lomet
Microsoft Research,
E-mail: jaedo@microsoft.com, lomet@microsoft.com

Ivan Luiz Picoli · Philippe Bonnet
IT University of Copenhagen,
E-mail: ivp@itu.dk, phbo@itu.dk

CR Subject Classification [500] Information systems DBMS engine architectures [500] Information systems Record and buffer management

1 Introduction

1.1 Problem and Opportunity

Cost/Performance Data should be placed at the most cost and performance-effective tier in the storage hierarchy. In the storage hierarchy, performance and cost decrease with distance from the CPU. We have argued earlier [16,33] that reducing I/O cost can have a substantial positive impact on the cost/performance. The cost of storage for data is always paid, while the cost of execution on it is paid only when the data is used. The cost equation then becomes:

$$Cost/sec = Storage\ cost/sec + (Ops\ executed/sec) \times Op\ Cost$$

This can be seen in Figure 1. Flash storage cost is lower compared with the storage cost of main memory (Figure 1.a), while its operating cost is higher due to the I/Os needed to bring data into main memory (Figure 1.b). Using this formulation, Figure 1.c illustrates the relative cost of executing operations on cached data versus data that resides on flash, as performance increases. We compare a main-memory system (where data is always in memory), SSD-based system (where data is on flash and only brought to memory when used) and data caching system that places data at the most cost/performance-effective tier in the storage hierarchy. As shown in the figure, a main-memory system is best when performance is high, an SSD-based system is best when performance is low, while a data caching system

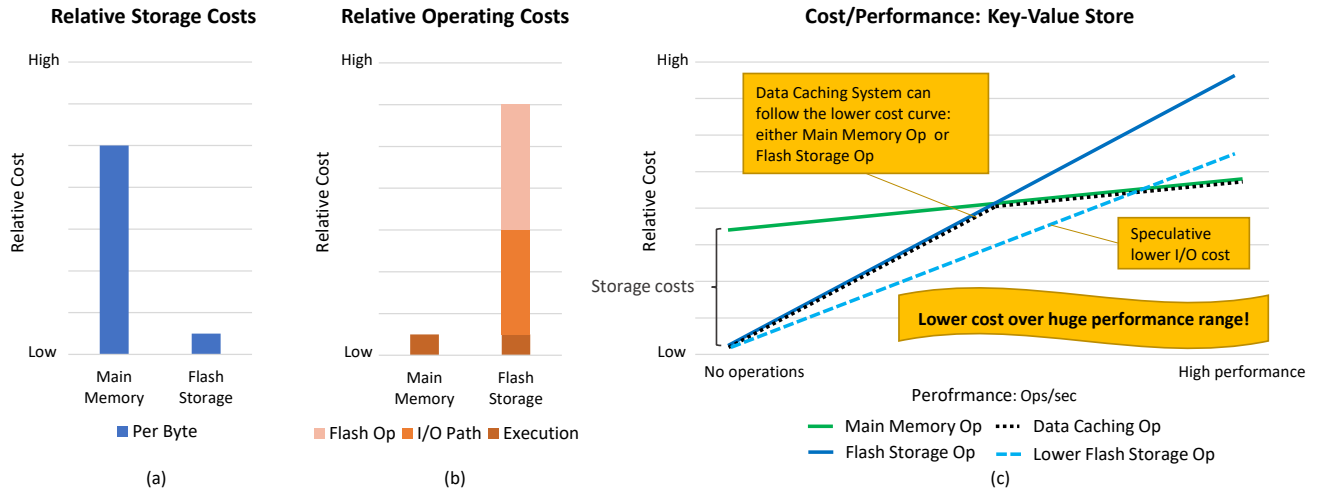

Fig. 1: It is all about relative costs! When data is in main memory and when it comes from an SSD, (a) flash storage cost is lower, but (b) execution cost is higher for reading the data into main memory. (c) cost vs performance for key-value store operations. The data caching system can follow the lower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

is best throughout the cost/performance space (i.e., the dotted black line). More importantly the cost over a large part of the performance range can be further lowered for SSD-based and data caching systems by reducing the I/O cost (i.e., the dotted light blue line).

Coping with I/O Flash-based solid state drives (SSDs) are sold as self-contained hardware units that support the standard block-at-a-time I/O interface for compatibility with the classic hard disk interface. A flash translation layer (FTL) [11], running on the SSD controller, enables logical in-place page update despite flash needing erases between writes. FTLs use log structuring (LS) virtualization [39] to avoid update-in-place at the flash storage layer, instead writing pages to new locations. LS is also used for file systems and cache managers to reduce host I/Os. The I/O path is expensive, and LS reduces its execution frequency. But LS needs garbage collection (GC) and checkpoint/recovery (CKPT/REC), which cuts into the performance gain. Further, with LS on a CPU using an SSD, both end up supporting LS functionality. We aim to reduce the cost of I/O by removing this redundancy.

1.2 Our Approach

Batching I/O We build a new FTL exploiting an open-channel SSD (OCSSD) that provides only read, write and erase functionality against raw flash memory. We augment the OCSSD with our own storage controller and implement the usual SSD FTL, with its GC and CKPT/REC. So, what changes? The primary advantage of implementing LS on the host is that many pages can be written in a batch, from a single buffer, with a single write I/O. GC and CKPT/REC are overheads. We modify the SSD block-at-a-time interface so that, from one buffer, we can write a disparate batch of pages (like LS) with one I/O. The host sends a buffer with a batch of data pages together with metadata describing which pages are included in the batch. By doing this, the host enjoys the benefits of LS while virtually eliminating its overhead.

If the hosts reads a logical page from SSD, it needs only provide the logical page id (`LPID`). The storage controller maps `LPID` to flash storage location, reads data from this flash location, and sends the resulting data back to the host. Reads can also be batched, with the host requesting a set of pages to read.

Moving LS Overhead to the SSD Controller With host-based LS, a conventional SSD receiving a write sees a storage start address and data length. It knows nothing about what the host sees as a collection of non-contiguous logical pages. The host maintains a mapping table associating an `LPID` for each page. Our new approach "outsources" the mapping responsibility to the SSD controller, giving it both data and `LPID` for each page in a multi-page buffer. Like an LS-based file system, we designate the first page of the I/O buffer as a metadata page containing the set of `LPIDs` for the pages contained in the buffer.

Now our storage controller maintains the mapping between `LPIDs` and physical locations. Hence, it is responsible for CKPT/REC to provide mapping table durability. Further, the controller can freely move pages around in flash storage. Hence, the controller must now

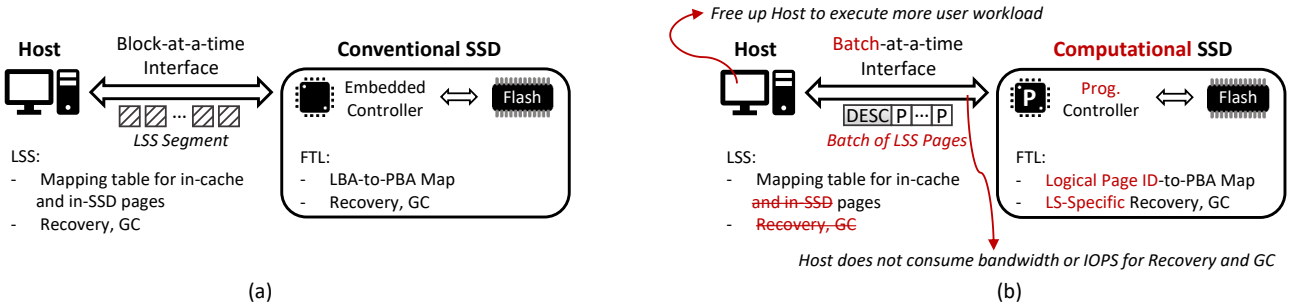

Fig. 2: (a) Host with a conventional, block-oriented SSD (b) Host using an SSD with an LS engineered controller that allows a batch interface. LSS and FTL denote Log-Structured Store and Flash Translation Layer, respectively.

also perform GC, moving pages as required to enable the erasure of flash blocks and their subsequent reuse.

Our hypothesis is that offloading GC and CKPT/REC from the CPU to the storage controller not only simplifies the host system, saving CPU I/O execution cycles, it is more efficient overall. We do not consume I/O bandwidth for GC or CKPT/REC. These functions are no longer replicated between host-LS and FTL, they are specialized in our SSD controller. Further, the CPU no longer needs to keep a mapping table in main memory that contains the flash addresses of its pages. Rather, it merely needs to know if a page is cached, and if not, it uses an `LPID` to request the associated data by way of the storage controller.

1.3 Our Contributions

We have transformed a data caching system implementing host-based LS on top of a traditional SSD, into a system where LS is implemented on the storage controller of a programmable SSD (see Figure 2). This transformation is based on the introduction of a new storage interface: batch I/O. Realizing the storage controller's modified FTL component that supports this new interface is not trivial. There are a number of concerns that software does not usually deal with:

- **Hardware errors** Traditionally, systems using storage devices do not worry much about hardware errors, assuming (mostly correctly) that the controller software deals with that. When we implement controller soft- ware, it becomes our responsibility to cope with errors in flash.
- **Wear Leveling** Controller software needs also to deal with wear leveling, counting erase cycles and provisioning storage so as to balance the erases across erase blocks.
- **Durability** Before the FTL acknowledges a write, it must ensure durability for the data being written and the updates to the mapping table. The FTL must ensure that information needed for recovery can be accessed and processed to ensure high performance recovery and availability.

Implementing and evaluating a batch I/O storage interface is the overarching contribution of this paper. Batch I/O reduces the cost of I/O while improving the effective IOPS rate. We provide a background on SSDs and log structuring in Section 2. We introduce the batch I/O storage interface in Section 3, another contribution. Here we need to balance write performance, latency and concurrency with user needs and understandability. Section 3 also provides an overview of our log structured SSD store. This sets the stage for how we implement the batching functionality.

- **Durability** Section 4 describes how we use database style logging and recovery to provide durability in the FTL. Our contribution here is in how we deal with media failures during normal operation, in writing both pages and log records.
- **Garbage Collection** Section 5 deals with garbage collection (called cleaning in log-structured file systems (LFS) [39]). Our contribution here is in exploiting an out-of-band controller accessible metadata region to track free(over-written) pages.
- **Checkpoint** We describe how we decompose SSD controller state that needs to be checkpointed in section 6. Imposing a hierarchy on resources being checkpointed is not entirely new, but its systematic exploitation is a contribution.

The OX Framework constitutes the software foundation and the basic functionality for our FTL. We detail how it was modified to efficiently support the batch I/O storage interface in section 7. Working close to the hardware is substantially more arduous than at higher levels of the system stack. A final contribution is our evaluation of the resulting performance in Section 8. The results demonstrate the effectiveness of our effort. We discuss related work in Section 9, and end with a short conclusion in Section 10.

2 Background

2.1 Solid State Drives (SSDs)

A modern SSD is split into two components [13], storage media and controller. The storage media is composed of arrays of flash chips wired in parallel on physical channels. A flash chip consists of multiple blocks, each of which holds multiple pages. Each page is further decomposed into fixed-size sectors with an additional out-of-bound (OOB) area (`TAG`) that is primarily used for Error Correction Code (ECC) and user-specific data. The unit of erasure (`EBLOCK`) is a block while writes are done at the granularity of pages (`WBLOCK`), and reads can be performed at the sector unit (`RBLOCK`). In the remainder of this paper, we will frequently refer to flash memory elements. Table 1 provides the terms we will use subsequently.

The controller is responsible for communicating to a host via the traditional block-at-a-time interface, and is designed to effectively manage the underlying storage media where in-place updates are not allowed. The controller implements a form of log structuring [11] in a Flash Translation Layer (FTL) that maps a flat logical address space to the hierarchical physical address space available in the SSD. The FTL also implements garbage collection to reclaim flash blocks, possibly containing valid data, and wear leveling to ensure that flash blocks wear evenly to prolong SSD life.

One of the keys to the widespread adoption of SSDs in the enterprise was due to their support of the same block-at-a-time interface that hard disk drives (HDDs) have used for several decades. But at the same time, this led to sub-optimal storage utilization and performance [34, 22, 27]. To address this problem, the SSD industry is facing two emerging changes - exposing SSD internals, and offloading host processing to SSDs.

2.1.1 Trend 1: Exposing SSD internals

In conventional SSDs, the main role of the FTL embedded on the SSD controller is to hide the complexity of managing internal storage media. Alternatively, SSD's internal media geometry and parallelism can be exposed to a host by implementing FTL responsibilities such as logical to physical mapping and garbage collection on the host side. With a host-based FTL, the host can control data placement and I/O scheduling based on user requirements [18, 24]. Parts of the industry are moving in this direction: Open-Channel SSDs [9, 38], for example, introduces a new I/O interface, called *Physical Page Addressing* that enables the host to access physical flash pages, not possible with the traditional block device abstraction. In an open-source project called Project Denali [2], Microsoft proposes a model allowing software-defined data placement on the SSD. In early 2019, Alibaba announced the deployment of open-channel SSDs to their hyper-scale data centers [44].

2.1.2 Trend 2: Offloading host processing to SSDs

Modern SSDs package processing (e.g., storage controller) and storage components (e.g., DRAM, Flash) for routine tasks such as mapping and garbage collection. These computing resources present an opportunity to execute user-defined functions inside the SSDs [15, 26, 37], which has evolved from the pioneering idea of Jim Gray's active disks [19] to a new generation of SSDs allowing such in-situ processing, called "computational SSDs" [17]. Computational SSDs include general-purpose, multi-GHZ clock speed, multi-core processors with built-in hardware accelerators (e.g., compression and decompression [10], pattern matching [20], FPGA [35]) to offload compute-intensive tasks from the processors, multiple GBs of DRAM, and tens of independent flash channels to the underlying storage media, allowing GB/s of internal data throughput. In the year of 2019, the Storage Networking Industry Association (SNIA) set up a technical working group [6] to standardize device interoperability, management and security among SSD vendors developing a number of different technologies and approaches to computational SSDs.

2.2 Log Structuring

Log-structured file systems (LFS) [39] were originally designed to minimize random I/O overhead on HDDs. LFS batch random updates in a large main memory buffer first, and then sequentially write the whole buffer as a large segment to the HDD. This design maximizes system write throughput by exploiting sequential I/Os, but the system must maintain a persistent mapping table that stores the latest location for each write. More importantly, periodical garbage collection must be performed to reclaim disk space occupied by stale data to ensure contiguous free areas for sequential writes. Log structured techniques have been explored as a way for a host to exploit SSDs [28, 42].

Internally, SSDs use log structuring approaches in their own FTLs to overcome the absence of in-place updates on flash memory. Both host-based LSF and SSD FTL implement their own mapping, cleaning, and recovery mechanisms. These redundant functionalities result in unnecessary management overhead [25, 43, 41].

Term	Name	Size	Description
RBLOCK	read block	4KB	Smallest readable storage unit
WBLOCK	write block	32KB	Smallest writable storage unit
EBLOCK	erase block	8MB	Smallest erasable storage unit
TAG	out-of-bound area	16B per RBLOCK	Controller accessible metadata

Table 1: Flash memory terms

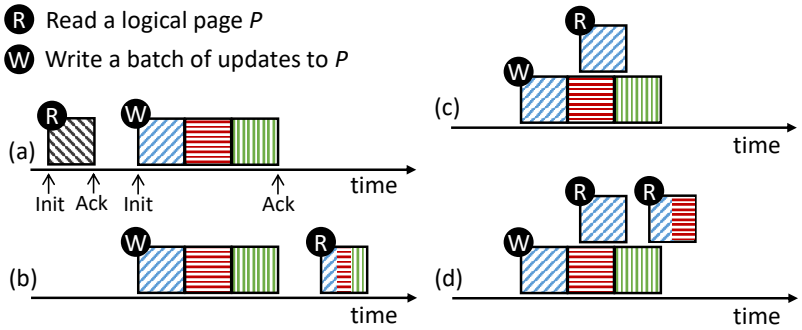

Fig. 3: Examples of concurrent read and write operations on a logical page P where blocks of different patterns indicate different states of P.

3 The Batch I/O Storage Interface

3.1 Batching multi-page writes

Log structuring replaces multiple write I/Os (one for each logical page (LPAGE)) with a single I/O (of a large buffer containing a collection of LPAGEs). Batching the pages enables sharing of a large part of the I/O path to the secondary storage device. As discussed in [33], this impacts both performance and cost of execution for operations requiring an I/O.

LPAGEs within a log structured buffer are time ordered, i.e., the updates to pages are intended to be in the order of the LPAGEs within a buffer, and between buffers. This order determines the visible states of updated blocks, and reads must expose SSD state as reflected in the write order as illustrated with examples in Figure 3:

(a) a read preceding a write batch (i.e., it is acknowledged before the write batch is initiated) sees a state that includes no updates in the batch.
(b) a read following a write batch (i.e., it is issued after the write batch is acknowledged) sees a state that includes all updates in the batch.
(c) a read that is concurrent with a write batch (i.e., it is initiated and/or acknowledged between the times a write batch is issued and acknowledged) sees a state that includes a time ordered prefix of the blocks in the batch.
(d) a second read B concurrent with a write batch but later than an earlier acknowledged read A sees a prefix of the blocks of the batch that includes the prefix seen by A.

Lastly, a write of a buffer is always completed atomically, meaning that all updates within a batch eventually appear in the data state on the SSD or none of them do. Put differently, the batch I/O interface offers transactional guarantees for write operations. It is a significant evolution compared to the traditional block I/O interface.

3.2 Log Structuring Functionality

3.2.1 Host-Based Log Structuring

Host-based log structuring requires a data management system, e.g., Deuteronomy, to assign "storage addresses" for its LPAGEs, identified with logical page ids (LPIDs). The host uses LPIDs when accessing secondary storage by first translating them into Logical Block Addresses (LBAs) on the SSD. The SSD knows nothing about this translation. It simply sees large blocks being written to sequential SSD locations. It uses the host based SSD addresses as LBAs, and maps them to the physical address space on flash. This logical-to-physical (L2P) table is fully hidden from the host. These mappings are illustrated in the Host-Based LS column of Figure 4.

Thus, the SSD, when garbage collecting, simply erases flash blocks and moves any live data it contains. This is hidden from the host. The host must do LPAGE level garbage collection, and thus read its previously written

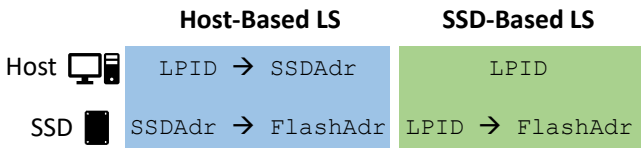

Fig. 4: The double mapping of host LPAGEs with their LPIDs to physical flash locations. Note that Host-Based LS and SSD-Based LS denote host-based and our SSD controller based log structuring, respectively.

buffers back into main memory to relocate still valid LPAGEs.

The host-based storage management must provide durabililty, maintaining the mapping between LPIDs and physical locations up to the last acknowledged write. Without recovery, a log structured store would either compromise this durability or require immediate durable mapping update. To speed up recovery, periodic checkpoints are necessary. Note that the SSD also implements recovery and checkpointing to guarantee the durability of the L2P mapping table.

3.2.2 SSD-Based Log Structuring

The SSD controller based log structured store accepts large, multi-page buffers. Unlike in a host based system, where the SSD knows nothing about the logical blocks within a buffer, the LPAGEs in each buffer are described via metadata (DESC) that resides in the first block of the buffer. DESC indicates the logical blocks that are present in the buffer and their offsets in the buffer.

This SSD-Based Log Structuring is in effect a FTL, specialized to support the batch I/O interface. As any FTL, its core functionalities are:

- maintaining a LPAGE-to-physical mapping table, that we denote MAP(see SSD-Based LS column in Figure 4);
- garbage collection based on detecting when an earlier write of an LPAGE has been over-written by a later write to the same LPAGE;
- recovery and checkpointing since MAP(which is the major recoverable element in the system) is under its management.

We describe recovery, garbage collection and checkpointing in the next sections. In the rest of this section, we describe the data structure we have devised to represent the mapping table in a memory efficient way.

MAP is organized as a hierarchical search structure as shown in Figure 5. We exploit this hierarchical structure for both cache management and incremental checkpoints. Since some LPAGEs may also be cached in the controller DRAM, each MAP entry (8-byte flash address) contains a flag indicating whether the LPAGE is cached or not. An associative side table (much smaller than MAP) is used to locate cached LPAGEs. This avoids a potential doubling of the size of MAP, were it to always include both flash and DRAM addresses. Depending on the SSD capacity, it would be not viable to entirely load MAP to the controller DRAM cache. As an instance, for a 2TB flash drive MAP's size is 4GB. Thus MAP is organized to be aligned with flash page boundaries so that parts of it can be flexibly cached, and large parts of it do not consume memory on the controller cache.

Pages of MAP are found via a smaller table MAP_Sthat tracks MAP pages in the same way that MAP tracks LPAGEs. Each entry of MAP_S includes not only a flash address, but also a cache address of a MAP page because MAP pages are likely to be cached in the controller cache. Even with 16 byte entries, the size of a complete MAP_S is just a few megabytes (in our example with the 2TB flash drive, MAP_S has a size of 16MB), which fits in contiguous locations of controller memory. An LPAGE is found by (1) using a 20 bit LPID prefix as an index to the MAP_S entry that points to the MAP page referencing the LPAGE; (2) accessing the MAP page to yield the flash pointer for the LPAGE. This two level indexing enables the controller to keep in cache only the MAP pages needed for active LPAGEs.

4 Durability and Recovery

Durability here refers to the need for the FTL to recover MAP and LPAGE states up to and including the last acknowledged buffer write prior to the system crash. This recovery step is not needed for conventional update-in-place storage systems.

4.1 Log Structuring Crash Recovery

MAP is treated as part of the SSD "database". Every update for an LPID updates the MAP database by storing a new physical flash address into its MAP entry. Should the system crash, we replay the SSD log to recover MAP, using LPAGE physical locations as the updated values for MAP LPID entries. While LPAGEs written by the host are the source for updates, the recoverable updates are the changes to MAP entries. There are other parts of our "SSD database" that we need to recover as well, and we discuss them below.

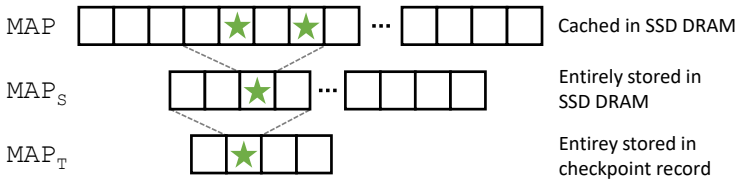

Fig. 5: The hierarchical structure used in managing the cached part of MAP and producing an incremental checkpoint for it.

4.2 Normal Operation

Recovery is tied to how we perform updates during normal operation, e.g., the database world uses write-ahead logging (WAL) [36]. WAL ensures that updates re stable on the log ahead of them being entered into the database. If a failure occurs, the log is replayed to recover database state.

Here the FTL itself provides durability. Whatever is written to a WBLOCK (See Table 1) is durable. Should an SSD error prevent the write from succeeding, the WBLOCK state can be determined by an attempt to read it. RBLOCKs each have a TAG field where we store the LPID associated with the RBLOCK. This makes recovery possible by scanning the disk and reading TAGs. But we want much faster recovery. So, instead of scanning the entire SSD, we replay a recovery log from the last checkpoint (see Section 6).

4.2.1 Update Protocol

Our recovery log is derived from the DESC block contained in each batch. A DESC associates LPIDs with buffer blocks. The FTL associates an LPID with the physical flash address at which we store its data. This is illustrated in Figure 6. The mapping metadata in DESC is used to create a redo recovery log block (RDESC) that is written to the SSD recovery log prior to the associated data blocks being written to the designated flash addresses. The ordering for writes in the RDESC is the same as the ordering of blocks in the write buffer. We incur latency for posting the RDESC in our WAL protocol as we must wait for an acknowledgement of success before we continue with the update the associated SSD LPAGE physical locations.

An RDESC does NOT enable redo of LPAGE writes as it does not contain LPAGE data. It only permits us to restore MAP. It assumes that LPAGEs have been successfully written to flash locations designated in the RDESC. RDESC permits us to test the flash locations by reading them at recovery time, in order to determine whether they have been successfully written.

4.2.2 Page Write Failures

An LPAGE write might fail. Should this happen, we need to write it to another location, prior to updating MAP. We also need to record on the log that the LPAGE is not where the original RDESC indicated. We use an **amendment** log record RDESC_A that provides a new location for any blocks whose writes have failed. The RDESC_A is organized like the RDESC but only has entries for LPIDs whose writes failed. RDESC_A is also written to the log prior to writing to flash the data blocks that it describes. If writes in the RDESC_A also fail, we continue writing RDESC_As until we have successfully written all blocks. Updating MAP does not start until all blocks are written successfully to flash storage. Once MAP is updated, completion is returned to the host system.

Given that LPAGE writes can fail, how does recovery know, after the loss of MAP in controller volatile state, whether a set of LPAGE updates has been installed? We can verify each write in recovery log RDESCs. However, this is expensive, requiring reading each physical address to determine whether the write succeeded. To avoid this and to speed up recovery, we write a **done** record for a RDESC when its updates are completely installed. A recovery log RDESC with a **done** record is confirmed to be durable without further checking. Also we do not wait for the RDESC **done** record to be durable before acknowledging the writing of DESC data blocks.

4.3 The Recovery Log

Recovery logs are usually assumed to be both logically and physically sequential on durable storage. But we want to make RDESCs for the log durable promptly, since writing RDESCs is in the latency path of a host I/O operation. Further, SSD storage is partitioned in various ways and is consumed and re-cycled in relatively divergent ways. So we want to construct a logically sequential log which is not required to be physically sequential in flash storage. We do this by means of a doubly linked list.

Each RDESC has a back link that points back to its predecessor block, and a forward link that points to

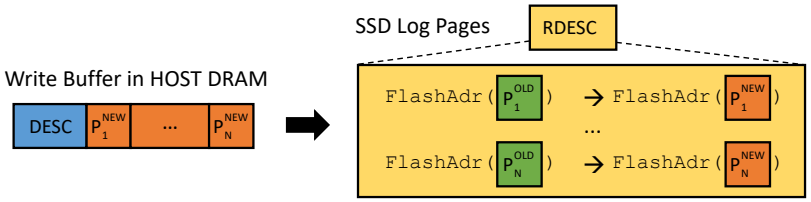

Fig. 6: Transformation of DESC describing blocks in a write buffer identified by LPIDs to a redo log recovery block RDESC that describes where the previous version of data for each LPID has been written on flash, and where the new version would be written.

the location of the successor block. This permits the recovery log to be traversed both forward and backwards, without being physically sequential. However, this is complicated by the fact that log writes can also fail. For this reason, we include an ordered vector (size specified by a parameter) of forward links with each RDESC. A log write failure triggers a retry at the next forward link of the vector.

In an RDESC, in addition to the new locations where updated data blocks are to be written, we include the prior location of each updated data block. This becomes an undo and redo log. We do not expect to use the undo information to roll back MAP, even for unfinished buffer writes because MAP is not updated until all data blocks are durable. Rather, undo information permits us to recover garbage collection information using the same log (see next section).

5 Garbage Collection

Flash memory requires that a block must be erased (EBLOCK) before the pages it contains can be written. This is why SSDs use log structuring for their FTLs. During updates, new data is written to a newly allocated block while old versions of the data "hang around" until garbage collected.

5.1 Erase Block Based Garbage Collection

The host multi-page write buffer is completely independent of the geometry of the SSD. To exploit SSD controller parallelism, pages of the write buffer are written across multiple EBLOCKs, in write block (WBLOCK) units. However, an EBLOCK, is the unit of garbage collection as it is the unit of erasure. Thus, we need to determine, per EBLOCK, the physical pages that have been over-written by subsequent updates to their contained LPAGEs. These over-written RBLOCKs then need garbage collection to reclaim their space. This is done by moving (re-writing) the still current RBLOCKs of an EBLOCK **old** to a new location in another EBLOCK **new** that was previously erased, and is thus writable. Once this has been done, all RBLOCKs in **old** have been "over-written" and **old** can be erased and re-used.

We need metadata associated with an EBLOCK, outside of the physical RBLOCKs to which logical pages are mapped, in order to track

- which RBLOCKs are garbage and which are not;
- how much storage (in RBLOCK units) is garbage in an EBLOCK;
- which logical pages that are not garbage need their RBLOCKs to be relocated to another EBLOCK and their MAP entry updated.

We maintain a global bit vector GBITV to identify which RBLOCKs are garbage and which are not. The segment of GBITV associated with an EBLOCK also permits us to determine how much EBLOCK storage is garbage. We schedule EBLOCKs for garbage collection in available storage (amount of garbage) order. Reclaiming this storage is the payoff for doing garbage collection.

We use an EBLOCK's GBITV segment to identify still valid pages (RBLOCKs) in the victim EBLOCK, i.e., the EBLOCK being garbage collected. The garbage collector reads each valid RBLOCK and its associated TAG field (See Table 1) that tells us the LPID for the corresponding MAP entry pointing to the RBLOCK. Then the garbage collector updates this MAP entry to the new RBLOCK containing the LPID page's state. This approach, exploiting TAG metadata that is not visible to the host, permits the controller to clean an EBLOCK without reading the entire EBLOCK into controller memory.

We have implemented a very simple scheme for separating hot data from cold data, a technique found to improve the garbage collection efficiency of log structured file systems [39] (i.e., reducing the RBLOCKs needing to be re-written per RBLOCK of storage reclaimed). This reduces the re-writing of cold data, and focuses most garbage collection on the current (presumed hot) data. The garbage collector uses a set of EBLOCKs for re-writing garbage collected RBLOCKs that is different from those used for ordinary user updates. There are

other more sophisticated strategies that can improve garbage collection efficiency further [39].

5.2 Protecting the Recovery Log

Section 4.3 described the recovery log as a linked list of RDESCs. But RDESCs are around the size of an RBLOCK, not a WBLOCK. We fill the rest of the WBLOCK containing an RDESC with data pages. This works well by permitting us to fill more fully each WBLOCK containing an RDESC, both reducing wasted space while maintaining low latency for the WAL protocol.

However, our garbage collector now has a problem if it chooses to garbage collect an EBLOCK containing an RDESC. We do not have a simple way of relocating the RDESCs of our log. The problem here is with maintaining the integrity of the log's linked list. Various indirect mappings using a relocation table are possible but awkward. Instead, we mark an EBLOCK as NOT subject to garbage collection until our checkpoint process can truncate the part of the recovery log that is in the EBLOCK.

6 Checkpointing

Checkpoints capture the state of a system at a given point in time. They are used to truncate the recovery log and thus speed up recovery in the event of system crash. Should our SSD controller crash, we apply redo log records (Section 4) from the latest complete checkpoint to the latest completed **done** record to restore the SSD state.

6.1 Checkpointing Strategy

We use a fuzzy incremental checkpoint that is an application of techniques used in database systems, e.g. [36] to truncate the recovery log.

- The checkpoint is fuzzy when it does not capture the system state at a precise point in time. Rather, it captures the state over some time interval, and uses redo recovery to make the state precise as of the end of the interval.
- The checkpoint is incremental when it does not capture the entire system state. Rather, it captures the part of the state that has changed since a prior checkpoint and merges that with the unchanged part.

We use this fuzzy incremental checkpoint to capture the state of MAP. In addition, we need to capture the states of the garbage (stale data, written and not yet reclaimed), and the flash block wear counters used for wear leveling. A checkpoint contains additional state, e.g. where the recovery log endpoint was when the checkpoint began.

6.2 Checkpointing Elements

We show the elements of the state that we capture for a checkpoint in Table 2. We introduce hierarchies to enable the states of MAP, GBITV, and WEAR to be saved incrementally. We save these states by treating their pages the same way that we handle data updates. That is, following the WAL protocol, we first write a recovery log record describing the metadata pages we intend to write, then we write their pages to the SSD as data. We accumulate enough state to properly utilize each recovery log WBLOCK. Once we have finished writing all changes to flash, we complete the checkpoint by writing the checkpoint record in a "well-known location" (see subsection 7.2.7).

6.2.1 Mapping Information

MAP updates are produced whenever a data block is written to flash storage. The update permits MAP to track the most recent state of any page on the SSD. This marks the MAP page as "dirty" in its MAP_S entry, meaning the cached version is an update to the version of the MAP page stored durably in flash memory. At some point, our checkpoint process needs to make the updated (dirty) MAP page itself durable. When this happens, the page in MAP_S that references this part of MAP is likewise marked as dirty and will need to be made durable.

We introduce an additional level for the mapping information, called MAP_T (T for tiny, see Figure 5) table because we do not want to write the entire MAP_S state to flash during a checkpoint. MAP_T has 16 byte entries <flash address, cache address> referencing the pages of MAP_S, and is 64KB in size for a 2TB flash drive.

When we persist MAP_S pages that have changed, these update entries are referenced in MAP_T. MAP_T is not on the access path to data blocks. Accesses all start at MAP_S. However, MAP_T permits us to unite the unchanged parts of the mapping information (referenced by entries pointing to unchanged parts of MAP_S) with the changed pages that we need to capture in our checkpoint. This allows us to incrementally save only dirty pages of MAP_S.

One way to view the above is that our mapping hierarchy permits us to treat pages in MAP_S and MAP as blocks of data whose writes are captured on the log. That is, every time a page in MAP_S or MAP is written

Term	Description
MAP	Mapping Table: LPID to physical flash address
MAP_S	Small table paginating MAP
MAP_T	Tiny table paginating MAP_S
GBITV	Bit vector: RBLOCKs containing overwritten state
$GBITV_S$	Small table paginating GBITV
$GBITV_T$	Tiny table paginating $GBITV_S$
WEAR	Wear Table: Erase cycles for each EBLOCK
$WEAR_S$	Small table paginating WEAR

Table 2: Recoverable state terms

to flash, we update the mapping hierarchy to reflect that change, and log that update in the same way that we log data block updates, but of course with different metadata to identify the metadata page being written.

6.2.2 Garbage State

Garbage is tracked using a bit vector per EBLOCK, each bit representing the state of a RBLOCK page. The cumulative size of the bit vectors (called GBITV) is 64MB for a 2TB flash drive, large enough that we want to capture only the parts that have changed, as we did with MAP. The smaller table that tracks the updated pages of GBITV is called $GBITV_S$ and is 256KB. $GBITV_S$'s size is not small enough to conveniently store entirely in our checkpoint record, so we introduce a tiny table $GBITV_T$, of size 512, as we did with MAP_T before for mapping data. $GBITV_T$ is stored in its entirety in the checkpoint record.

6.2.3 Erase Count

Each EBLOCK (the unit of erasure) has a limited number of erase cycles. We count and maintain a 4 byte counter per EBLOCK to track wear (the number of erase cycles) and store it in the WEAR vector. Total size of WEAR is 1MB. As we did with MAP, we paginate WEAR and only save the part of WEAR that has changed, by introducing $WEAR_S$, a page based index for WEAR. $WEAR_S$ size is 2KB and hence is small enough to be stored directly into our checkpoint record, without a tiny wear state vector.

7 SSD Controller Programming

Traditionally SSD storage controllers are programmed by storage vendors. The FTL is a proprietary firmware, that developers of host-based system do not have access to. Thus developers are stuck with a device whose functionality and capabilities are outside their control, even as many of them are modifiable in software. The advent of the Open-channel SSD provides an opportunity to break this state-of-affairs by decoupling front-end (FTL) and back-end (storage media) SSD management.

However, how to program SSD controllers is not obvious. As a way for offloading host processing to a storage controller, for example, application-specific codes can be executed on the top of a generic block-oriented FTL (e.g., Scaleflux [5], NGD Systems [3]). While this approach naturally improves performance by leveraging resources available inside the SSD (Section 2.1.2), we might miss a number of optimization opportunities across several layers on the data path in the FTL [9]. Thus instead of simply adding functionality on the top of a generic FTL, we want to map commands defined on an application-specific address space onto storage primitives defined over a physical address space. To do this, we must be able to design a new FTL.

7.1 The OX Framework

Recently, many robust open-source FTLs have been released. Pblk [9], for example, implements a full-fledged, host-based FTL exposing a traditional block I/O interface, and is released as part of the Linux Kernel 4.12. Intel released a user-space FTL in the context of SPDK [7]. Those FTLs, however, must remain generic. They are not meant to be modified to support application-specific address space. As a result, we designed and and developed a modularized framework (called OX) that can readily be modified to program storage controllers [38].

OX uses an abstract execution model with well-defined APIs, consisting of three layers (Transport and Parser, FTL, and Media Manager) between the host and the storage media as shown in Figure 7. A unique merit of modern SSDs is their rich internal parallelism, which is naturally abstracted through pairs of submission and completion queues. The three layers in OX also rely on a decoupling of command submission and notification of completion. The layers are asynchronously intercon-

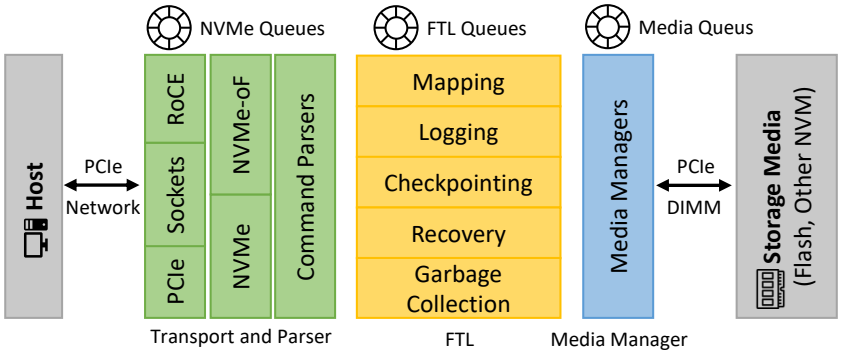

Fig. 7: Abstract execution model of the OX framework

nected with queue pairs. The number of queues in each layer can be customized depending on many factors including the number of available cores in the controller or the number of channels connected to the controller.

7.1.1 Transport and Parser

This layer is responsible for interacting with the host and for parsing commands with the goal of delivering reliable and consistent messages to the next layer. OX supports PCI express (PCIe) and network fabrics transport types. Examples of transports include PCIe messages for PCIe and Sockets/RDMA over Converge Ethernet (RoCE) for network fabrics. For transport protocols, OX implements both NVMe and NVMe-oF [4]. The first byte of a NVMe command must contain an operation code, which allows us to define custom commands and thus support new storage interfaces (e.g., batch I/O interface, see Section 3).

Incoming commands are sent from the transport layer to a command parser. The command parser reads the operation code and the data pointer in the incoming command (which could be a host memory address for DMA/RMDA, or an offset within a packet for TCP/UDP), and sends a re-formatted command to the next layer.

7.1.2 FTL

The middle layer manages the mapping of logical to physical addresses, and the associated tasks such as logging, checkpointing, recovery and garbage collection. As a high level, an FTL is a block box that exposes FTL submission and completion queues. Submission queues process commands submitted by the upper layer, while completion queues use callbacks for completion. Physical I/O commands are submitted to the next layer when reading/writing data to storage media is required.

7.1.3 Media Manager

This layer contains a number of media manager instances to support several different forms of storage media. Each type of media requires an instance in OX, which exposes a geometry abstraction of the underlying media. A media manager receives commands from the upper layer, and moves data to/from hosts according to the commands. Note that the data might not be located on the memory buffer in the storage controller, rather on the host connected directly or via fabric. In this case, DMA or RDMA is required to avoid memory copies for bypassing the storage controller during data transfers to/from the media.

7.2 FTL Programming with OX

Figure 8 shows OX's FTL architecture (including dependencies across nine FTL components) as well as the data path for both controller and user generated data movement. Controller I/Os (represented by solid lines) are synchronously executed by several components. For example, garbage collection may read and write to media as data get invalid, and block and mapping metadata need to be persisted during the checkpoint process. User I/Os (represented by dashed lines), on the other hand, are performed asynchronously. Note that the controller cache is only used for caching writes, not reads. When a write is issued, it is first posted to the queue in the write cache, and immediately returned if a write-back caching mechanism is enabled. Otherwise, the write is completed only after persisting data on flash.

7.2.1 Bad Block Management

A media EBLOCK is considered bad if the data it contains is unrecoverable or if an erase command fails due to the limited number of erase cycles per each block.

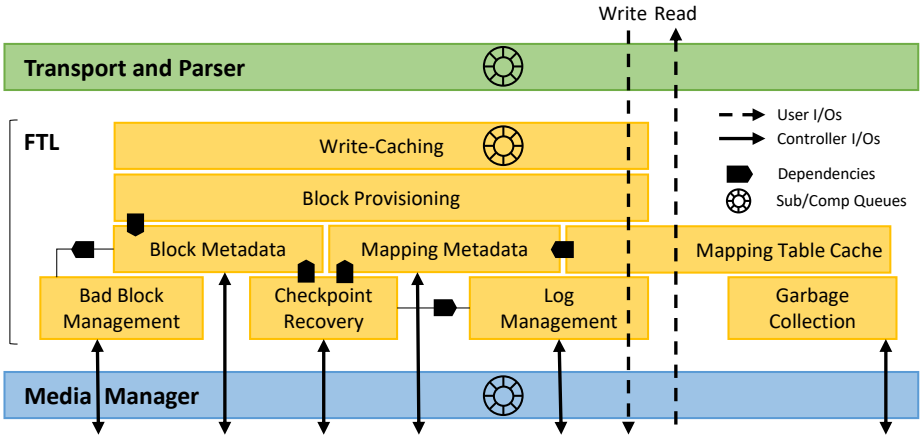

Fig. 8: Dependencies and interactions among nine OX FTL components.

For example, for single-level cell (SLC) flash it is typically 100,000 erase/write cycles, and for multi-level cell (MLC) it can go down to 3,000 - 10,000 cycles. Thus it is important to track bad blocks to remove them from the provisioning and get retired from any further usage. We maintain bad block information with an array of bytes representing the whole set of EBLOCKs in a device, which is stored on a dedicated set of EBLOCKs. For a 2TB 8-channel device, as an instance, each channel stores a 32KB bad-block table that is written to the first WBLOCK of the dedicated EBLOCK of this channel. When the table is updated, a new version is written at the next WBLOCK, and if the EBLOCK gets full, it is erased and the first WBLOCK is used again.

7.2.2 Block Metadata

We maintain metadata related to live EBLOCKs, which is used for components such as provisioning and garbage collection. The block metadata contains block types, numbers of erase cycles (i.e., WEAR in Table 2), and bit vectors representing invalid RBLOCKs (i.e., GBITV in Table 2). Each EBLOCK metadata has a 16-bit field that tells the status of this EBLOCKat any point of time. Status flags include: 1) *FREE* if EBLOCK is empty and ready to be written, 2) *FULL* if EBLOCK is full, 3) *OPEN* if EBLOCK is partially written 4) *HOT* if EBLOCK contains data written by users, 5) *COLD* if EBLOCK contains data moved by garbage collection, and 6) *META* if EBLOCK contains log data. The usage of each flag is described in the next section.

7.2.3 Block Provisioning

Block provisioning is responsible for data placement by providing physical addresses of EBLOCKs in a certain order to achieve full inter- and intra-channel parallelism on the SSD. At a high level, a write request is striped over multiple flash channels (that can be considered as independently operated I/O buses). Within a channel, the request is further pipelined to be simultaneously served by multiple flash chips.

At startup, EBLOCKs are grouped into three types per chip: *FREE*, *FULL*, and *OPEN*. Bad EBLOCKs are excluded from provisioning, and not added to any group. Then an *OPEN* EBLOCK from each chip is selected to create a fully intra-channel parallelized list called *line*. When physical addresses where data is written are requested, to maximize the inter-channel parallelism, a single EBLOCK from each line per channel is reserved, and then channels are selected in a round-robin fashion until the amount of requested addresses have been reserved.

Writes could be generated by several FTL processes such as user writes, garbage collection, checkpoint, and logging. Some of these processes require independent EBLOCKs where data is not mixed with other processes. We define three types (*HOT*, *COLD*, and *META*) of provisionings, each of which has its own *line*. By using several provisioning policies we avoid concurrency between write threads that could lead to wrong write sequence within an EBLOCK. Note that, as an exceptional case, and in order to maximize the space utilization, and to reduce the latency for the WAL protocol, EBLOCK could be filled up with both *HOT* and *META* data (see Section 5.2).

7.2.4 Mapping Metadata

Mapping logical to physical addresses is the essence of the FTL. Pairs of logical and physical addresses are placed sequentially to assemble a table (MAP, see Section 3.2.2) that represents the entire address space. Since the size of MAP, which depends upon the stor-

age capacity and the granularity of each entry, could grow tens of gigabytes, loading the entire MAP at once to the controller DRAM is not viable. Instead we split MAP into WBLOCK-size pages that are spread across flash chips and channels, and use a secondary table (MAP_S) to store physical addresses where MAP pages are written. Making MAP durable requires: (i) persisting dirty MAP pages that are cached, (ii) persisting dirty MAP_S pages, and (iii) sending a copy of an additional level of the mapping information (MAP_T, see Section 6.2.1) to a checkpoint. Once the checkpoint is completed, MAP gets durable.

7.2.5 Mapping Table Cache

Independent MAP pages that are aligned to the WBLOCK boundaries can be flexibly cached (and evicted when the cache is full), which allows us to efficiently access MAP entries while maintaining them in the controller DRAM. Multiple threads are allowed to access the cache, however, if a MAP page is being loaded or evicted, the working thread holds a lock to guarantee consistency. It also avoids the same page being loaded twice by concurrent threads. The cache component is not aware of durability, but responsible for updating MAP_S, which is small enough to be always cached to the controller DRAM, properly after loading or evicting pages. Note that, at startup, we do not warm up the cache by design, meaning that every first access to a MAP page requires a read from flash.

7.2.6 Log Management

A recovery log is used to ensure durability of metadata information. Even in the case of a clean shutdown, some mapping entries and block metadata may not contain the latest values due to the nature of the fuzzy checkpoint. The recovery log contains all changes performed after the latest durable checkpoint when a shutdown was caused by failure. For both cases, after a startup, we recover the metadata to a consistent state by applying updates that are recorded after the latest checkpoint. Logs, each of which is 64-byte in size, follows the structure described in Section 4.

7.2.7 Checkpoint and Recovery

The complete state that we need to maintain across system crashes, after having converted larger tables to updatable data, consists of MAP_T (516 bytes, Section 6.2.1), $GBITV_T$(512 bytes, Section 6.2.2), $WEAR_S$(2KB, Section 6.2.3), the current timestamp (8 bytes), the log location of where to start recovery using the recovery log (8 bytes), called the "redo scan start point" or RSSP, and the channel identifier (32 bytes) for the provisioning to continue the round-robin selection after recovery. This comprises our checkpoint record that we write to a "well known location" so that it can be found after a crash. Should the system crash, we first find the latest checkpoint record, and then execute recovery by reading the log from the RSSP until end of log, redoing the logged operations. End of log is detected when the next pointers in a recovery log block point only to erased blocks, not written blocks.

7.2.8 Garbage Collection

Our garbage collection design is based on a channel abstraction. If a channel contains *FREE* EBLOCKs less than a threshold, the garbage collector initiates the process by disabling the channel for further user writes (but user reads are allowed). Note that the garbage collector might need to wait until ongoing user writes on the channel are completed. Once the channel gets disabled, EBLOCKs in the channel are sorted based on the number of invalid RBLOCKs so that EBLOCKs that require less data movement could be recycled first. The remaining garbage collection process is described in Section 5.1.

In order to manage concurrency issues with user writes, the garbage collector uses the optimistic locking strategy [29] when updating MAP entries (after moving valid RBLOCKs to new locations). Let us suppose the garbage collector moves an LPAGE to a $RBLOCK_{gc}$, and at the same time, a new version of the LPAGE is written to a different location, $RBLOCK_{uw}$ by an user write. In this case, the user write has a higher priority over the garbage collector since the MAP entries always have to point to the latest versions. Thus, before updating the LPAGE's MAP entry with a standard fine-grained lock, the garbage collector reads the entry first. Then after holding the lock, it reads the entry again, and replaces the entry value only if those two entries with and without the lock are equal.

7.2.9 Write Caching

The write cache is an entry point for user writes, and is responsible for abstracting complex interactions with other FTL components required for persisting data on flash (Compared to this, user reads only involve the Mapping Table Cache component as they do not change the state of metadata.) Once a write buffer has arrived at the cache, LPAGEs are extracted by parsing the buffer, and then the LPAGEs are grouped into WBLOCK-aligned data blocks. As a next step, we request flash physical addresses, telling where the blocks will be written, from the Block Provisioning component, and store

logical addresses at the TAG fields of each block, which is useful for reverse mapping performed by garbage collection (Section 5.1). Finally we write the blocks to their designated flash addresses in parallel using multiple threads, which is the most expensive step in making the LPAGEs durable, and then update the MAP entries associated with the LPAGEs.

Writes are accepted in different granularities ranging from 4KB to several megabytes, depending on the underlying storage media. For hosts, a large logical write is seen as a single command that must be either successful or failed while such write is divided into multiple, WBLOCK aligned, and independent commands in SSDs. We guarantee atomicity (either all data belonging to the same logical write succeed or we abort them all) with the WAL protocol (Section 4.2). For each logical write with multiple LPAGEs, we append a log record describing old and new physical addresses of the LPAGEs. If a logical write fails, we do not update MAP entries nor append the **done** log record.

8 Evaluation

Our goal with the evaluation is twofold. First, we compare the impact of our new batch I/O storage interface (OXBatch) on the BwTree key-value store, compared to the traditional block I/0 interface. We thus focus our experimental study on a single-threaded version of the BwTree in order to isolate the performance impact of the storage interface. Second, we explore the performance impact of the different components of the log structuring solution that we offload to the SSD with OXBatch.

8.1 Experimental Setup

8.1.1 Hardware

We performed our evaluation via single-threaded experiments conducted on an Ubuntu Linux host machine with an Intel Xeon E5-1620 3.5GHz processor with 32GB of DRAM. Our computational SSD is built based on a standalone platform, named STT100, developed by Broadcom for developing storage applications. The platform uses a BCM5880X SoC equipped with an ARM Cortex-A72 1.8GHz processor and runs Ubuntu as a working OS. The storage media is a Westlake Open-Channel SSD from CNEX Labs attached to the STT100 via PCIe Gen3x8. Data transfer between the host and the computational SSD is performed via stream sockets with the NVMe-oF/TCP protocol. We configured the network speed between the host and the SSD to be 100Gbps, which guarantees that the network is not a performance bottleneck in all experiments.

8.1.2 Software

Figure 9 shows two system configurations used for our evaluation. In both configurations, the host is connected to a computational SSD.

The software system running on computational storage is OX. We consider two variants of OX: OXBlock and OXBatch. OXBlock[1] is a full-fledged, block-oriented FTL, in approximately 31,000 lines of C code. We ran the OXBlock inside the computational SSD (Figure 9a - Comp. SSD) to enable processing read and write operations via the standard block-based interface.

OXBatch (Figure 9b - Comp. SSD) is a new FTL that implements the batch I/O storage interface. This implementation relies on in-SSD log structuring with the following two new commands: (1) $\texttt{read}_{\texttt{LPID}}$ for reading an LPAGE with its LPID, which differs from the standard block read (used in the OXBlock) that required the starting address of the LPAGE, (2) $\texttt{flush}_{\texttt{batch}}$ for flushing a batch of LPAGEs entirely. Compared with the standard write of an array of bytes, identifies the LPAGEs by parsing the batch with its metadata (DESC). Note that the modularized OX framework, and the fixed-size (4KB) LPAGEs that are aligned with the OXBlock mapping granularity allows to reuse most of the OXBlock FTL components when building the OXBatch.

The software system running on the host is a key-value store based on BwTree [32], a latch-free, B-tree style index layered on LLAMA [30], a subsystem providing both log-structured cache and storage management. The original BwTree avoids updating base pages directly by storing modifications to a node in a delta record, and maintains a chain of such records for every page in the index, which is periodically consolidated into the base pages. We modified the original BwTree to perform updates in-place without creating delta chains, and therefore to process only fixed-size pages during the entire evaluation (Figure 9a - Host). We configured the BwTree page size to 4KB and write buffer size to 1MB.

We modified or removed the following components from BwTree with fixed-size pages to use the batch I/O interface with the LS-engineered SSD (as illustrated in Figure 9b - Host):

– The BwTree no longer needs to remember where LPAGEs are located on the SSD.

[1] The source code of OXBlock is available at https://github.com/DFC-OpenSource/ox-ctrl

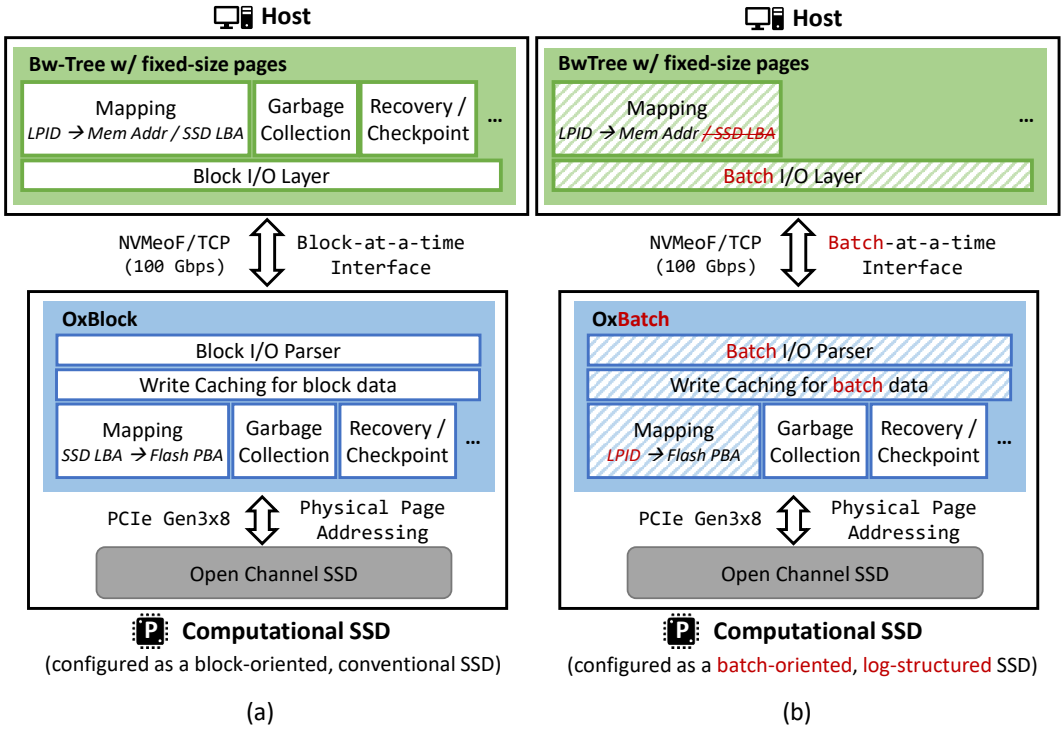

Fig. 9: Two experimental configurations with the OX running on a computational SSD. (a) BwTree_Block: host-based log structuring built using a block-based SSD, (b) BwTree_Batch: host batch I/O interface to log structuring built into an SSD.

– Checkpointing and recovering BwTree's mapping table is not needed as only cached LPAGEs are mapped to their main memory locations.
– Garbage collection is performed by the SSD controller being aware of LPAGEs' physical locations.
– BwTree's I/O layer needs to support the batch-oriented operations.

We experimentally set the size of the I/O queue depth used in the LLAMA to be large enough that the queue is never starved, and multiple asynchronous read and write requests can be issued simultaneously.

8.1.3 Workloads

We ran all experiments with a set of YCSB [12] benchmarks, which are a widely used framework for evaluating performance of NoSQL stores. Our YCSB dataset has 10 million unique records, each consisting of an 8-byte key, and a 100-byte payload. With this dataset, we evaluated the throughput performance of the system with two YCSB workloads, both of which contains 10 million operations: (1) a read-heavy workload with 95% reads and 5% updates, and (2) a write-heavy workload with 5% reads and 95% updates. For all workloads, keys to read or update are selected randomly from the set of existing keys in the index according to a Zipfian distribution, which has skewed access patterns common to NoSQL workloads.

All the results presented here are for cold experiments. For each run of benchmarks we newly initialized an index with records in the dataset, and then ran the specified workload for 300 seconds. We reported the total number of operations completed in that time, where operations are either reads or updates. To simulate real-time usage, the read and update operations are interleaved. Specifically, for the read-heavy workload, we performed 19 reads, then 1 update, then repeated the cycle; For the write-heavy workload, we performed 19 updates, then 1 read, then repeated the cycle.

8.2 Results

In this section, we present the overall throughput achieved for the YCSB workloads when the BwTree runs with a computational SSD (1) configured as a block-oriented SSD (BwTree_Block), and (2) configured as a batch-oriented SSD (BwTree_Batch) as shown in Figure 11. To better understand the impact of checkpointing and garbage collection, we enabled those features one-by-

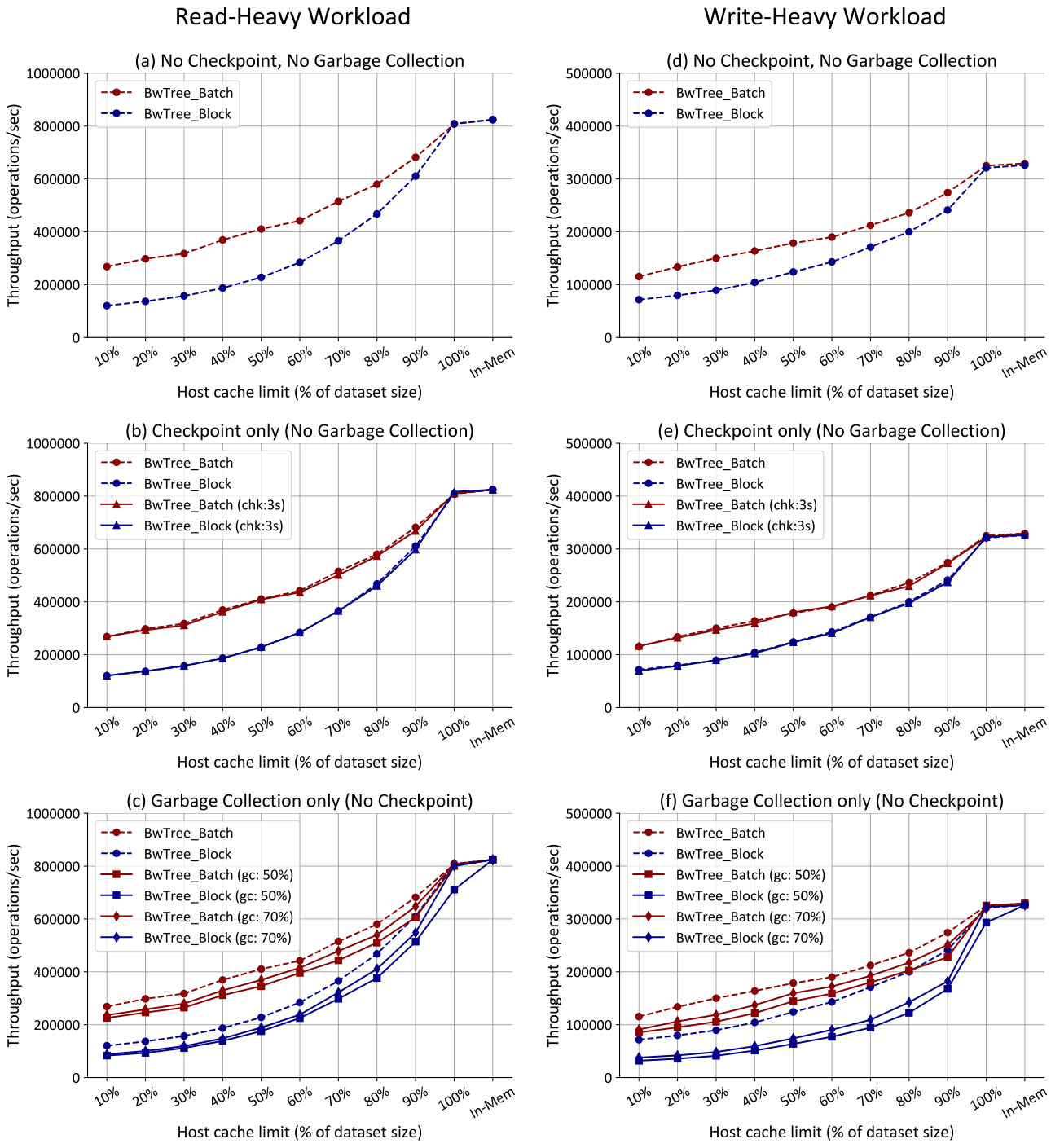

Fig. 11: Throughput of `OXBlock` and `OXBatch`, respectively when running YCSB benchmarks with read- and write-heavy workloads (a) without Checkpoint/GC, (b) with Checkpoint only, and (3) with GC only

one in a single-threaded environment. We varied the host DRAM cache size over a range measured as a percentage of the dataset size. In addition, we measured a pure in-memory throughput (`In-Mem`) where no read and write operations are issued to the SSD during the benchmarks. In contrast, when the host cache limit is set to 100%, all read operations are responded with data cached in memory while new or updated pages are (sequentially) flushed to the SSD whenever a host write buffer gets full.

8.2.1 Non-Durable Baseline

In this first experiment, we evaluated the throughput of the `BwTree_Batch` and the `BwTree_Block` in a non-durable setup where both checkpointing and garbage collection are disabled. The results are shown in Figure 11a and Figure 11d for read- and write-heavy workloads, respectively. As expected, the performance of both configurations slowed down with limited memory because of increased number of I/Os issued to the SSD, but reached the in-memory performance level as the entire dataset fits into memory.

When the dataset is larger than memory we observed clear benefits of using the batch-at-a-time interface compared with the block-at-a-time protocol. For example, as shown in Figure 11a and Figure 11d, `BwTree_Ba` outperformed `BwTree_Block` by 1.11 – 1.91× and 1.14 – 1.68× for read- and write-heavy workloads, respectively. The main reason for this behavior is due to the different write granularities that are supported by each configuration. Unlike the read path of `BwTree` where a single page is read at a time, in the write path, an 1MB-sized write buffer is flushed (when it gets full) to the SSD during the benchmarks. Once the flush operation starts, the 1MB data is first split into 17 packets[2] according to the NVMe-oF/TCP protocol (See Figure 9), and then the packets are sent to the SSD.

On the SSD side, `BwTree_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, `BwTree_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 `BwTree_Block` has to process about 17× more internal writes than `BwTree_Batch`, resulting in much more "done" log records (Section 4.2.2) that need to be generated and flushed before completing the 1MB write request due to the WAL protocol (See Figure 12). In addition, the small write granularity of `BwTree_Block` prevents fully exploiting the internal SSD parallelism - the maximum size of an internal write of `BwTree_Block` is bounded by the packet size, which is not big enough to leverage all flash channels at once.

8.2.2 Checkpoint

We next added a checkpoint process during the benchmarks (except the `In-Mem` setup) to study its impact on the throughput performance. In this experiment, at a given interval, a checkpoint process described in Section 6 occurs by the SSD controller for both the `BwTree_Block` and the `BwTree_Batch`. In addition to the SSD-based checkpointing, the `BwTree_Block` needs to trigger a host-based checkpoint to persist its `BwTree` mapping table entries pointing the pages stored on the SSD (i.e., in the `BwTree_Block` case, two checkpointing processes, one by the host and the other one by the SSD might need to be performed simultaneously.)

Interestingly, as shown in Figure 11b and Figure 11e, even in the case where a checkpoint is triggered very aggressively (e.g., 3 seconds), we observed almost no degradation on performance for both the read- and write-heavy workloads. This is mainly because the amount of state required to be stored during a checkpoint, which primarily depends on the size of the dataset, does not put enough pressure on the bandwidth available on the SSD in our experiment. For example, when the cache limit was configured to be 50% of the dataset, for the write-heavy workload (Figure 11e), the `BwTree_Batch` required to persist about 15MB metadata during a checkpoint[3], which does not incur much overhead to the

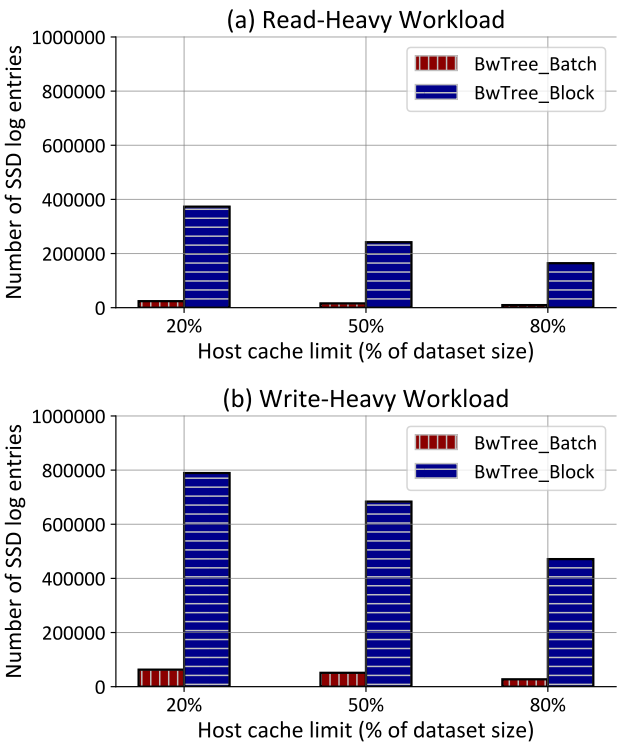

Fig. 12: Number of "done" log records (Section 4.2.2) generated for the durability of host write requests.

[2] 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.

[3] Due to the host-based checkpoint for persisting `BwTree` mapping table entries, the `BwTree_Block` required to write additional 3.5MB data during the checkpoint.

Cache Limit	Configuration	User Read	User Write	GC Read	GC Write	Reclaimed Space
20%	BwTree_Block	19.96 GB	28.08 GB	28.78 GB	3.74 GB	25.97 GB
	BwTree_Batch	45.89 GB	57.28 GB	22.93 GB	7.54 GB	56.86 GB
50%	BwTree_Block	13.16 GB	22.99 GB	23.48 GB	3.58 GB	19.90 GB
	BwTree_Batch	25.13 GB	40.96 GB	17.07 GB	5.21 GB	42.16 GB
80%	BwTree_Block	6.09 GB	17.58 GB	17.37 GB	2.83 GB	14.07 GB
	BwTree_Batch	10.80 GB	25.74 GB	9.84 GB	3.86 GB	25.44 GB

Table 3: The amount of data read from, or written to, the SSD for user requests and GC processing. The "Reclaimed Space" column shows the the amount of SSD space cleaned by GC.

Cache Limit	Configuration	Read Amplification	Write Amplification	GC Overhead
20%	BwTree_Block	2.442	1.133	1.252
	BwTree_Batch	1.500	1.132	0.536
50%	BwTree_Block	2.784	1.156	1.359
	BwTree_Batch	1.679	1.127	0.528
80%	BwTree_Block	3.850	1.161	1.435
	BwTree_Batch	1.911	1.150	0.538

Table 4: Read/Write amplifications and GC overhead where lower values are better.

available SSD bandwidth that is in the range of 1 – 2 GB/sec. We might see a noticeable performance drop with a bigger dataset, which needs to flush larger amount of data during a checkpoint, but a further study of this case is left as future work.

8.2.3 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 in progress, contributing to maximize 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, we examined the impact of the aggressiveness of GC on the workload throughput with two threshold options: a) aggressive threshold: 50%, and b) non-aggressive threshold: 70%, meaning that the GC process starts when the SSD space is 50% and 70% full, respectively. When such GC condition is met, the in-SSD GC process described in Section 5 is conducted by the SSD controller for both the BwTree_Block and the BwTree_Batch.

The BwTree_Block requires an additional host-based GC process initiated by LLAMA [30] that continuously reclaims space occupied by stale data to ensure continuous free areas for the appending of new versions of pages. Since versions of pages have different lifetimes, from the host-based log structuring perspective, very old parts of the log could contain current page states. To reuse this old section of the log, the still current pages states 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 LLAMA-managed GC simplifies BwTree_Block GC. Indeed, from BwTree_Block's perspective, it is trivial to identify victim blocks containing only obsolete pages. They correspond to the tail of the log managed by LLAMA.

Figure 11c shows throughput results of the read-heavy workload when enabling GC with the aggressive or non-aggressive GC options. As expected, compared with the case where GC is completely disabled (i.e., dotted lines) we observed performance degradation with the non-aggressive GC (gc: 70%) in the range of 5% to 12% for the BwTree_Batch (i.e., read lines with diamond marker) and 10% to 27% for the BwTree_Block (i.e., blue lines with diamond marker), respectively[4]. The performance drop is 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.

Compared with the read-heavy workload, the write-heavy workload shown in Figure 11f requires more space to be reclaimed at the cost of using processing power and I/O bandwidth that would be leveraged for user re-

[4] In some cases where the entire dataset fits into memory, no performance drops were observed as the SSD usage during the experiment was not big enough to meet the given GC condition.

quests, resulting in more significant performance drops (i.e., 8% – 22% drops for the BwTree_Batch and 24% – 47% drops for the BwTree_Block, respectively, with the non-aggressive option). Note that the aggressive GC (gc: 50%, lines with square marker) that cleans up orphaned pages more frequently than the non-aggressive case requires more space to be freed, resulting in about 5% – 10% further performance drops.

To better understand the impact of GC, we measured the amount of data read from, or written to, the storage media for (i) user requests and (ii) GC processing. We also measured the amount of SSD space reclaimed by GC. An example of such measurement when running the benchmark with the write-heavy workload by enabling the aggressive GC option is presented in Table 3[5]. Note that since the BwTree_Batch achieved much higher throughput than the BwTree_Block (Figure 11.f), it is obvious that a larger amount of data needed to be transferred to process the user requests (e.g., 28.08 GB vs. 57.28 GB written to the SSD by user requests when the cache limit was set to 20%). Since more data was written to the SSD, it is clear that the BwTree_Batch required to trigger GC more frequently, reclaiming larger space (e.g., 25.97 GB vs. 56.86 GB reclaimed by GC when the cache limit was configured to 20%).

Table 4 presents the results of analyzing the aggressive-GC efficiency for the write-heavy workload in terms of read (write) amplification and GC overhead. Due to the GC activity, the actual amount of data that is read or written physically to flash is greater than the logical amount that is intended to be read or written. This undesirable phenomenon is called "Read(Write) Amplification". A lower amplification is always desirable. In addition, we measured the GC overhead representing the amount of data GC moved per unit of reclaimed space. i.e., (*GC Read+GC Write*) / *Reclaimed Space*. We use the following simple formula [23] to calculate read(write) amplification.

$$Read(Write)\ Amplification = \frac{User\ Read(Write) + GC\ Read(Write)}{User\ Read(Write)}$$

As can be seen in the table, BwTree_Batch achieves about 1.5X – 2X lower read amplification and about 2.5X lower GC overhead compared to LLAMA GC on top of BwTree_Block.

[5] Note that we found similar observations when running the read-heavy workload.

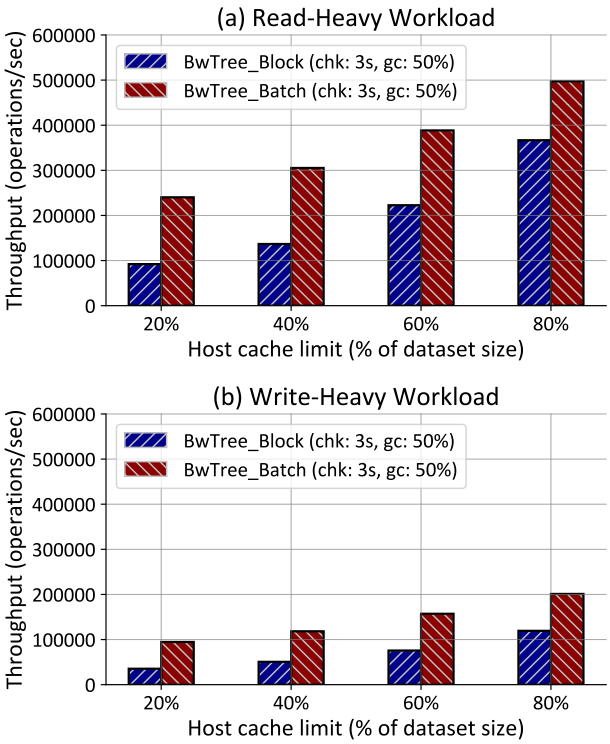

Fig. 13: Overall end-to-end throughput results with both checkpointing and GC are enabled.

8.2.4 End-to-End Throughput

Figure 13 presents the overall throughput results with both checkpointing and GC enabled. For the read-heavy workload, BwTree_Blockhas a throughput of 92K (20% cache limit) – 366K (80% cache limit) ops/sec while BwTree_Block has a throughput of 240K – 497K ops/sec, representing a 1.4× – 2.6× speedup depending on the available host cache size (see Figure 13a). A similar speedup is observed for the write-heavy workload shown in Figure 13b - 35K – 119K ops/sec for the BwTree_Block and 95K – 201K ops/sec for the BwTree_Batch, respectively.

8.3 Discussions

8.3.1 Comparison with Conventional SSD

As a sanity check, we compare the performance of our solutions based on OX, with a commercial NVMe SSD. The results are shown in Figure 14. The dotted green curve represents throughput for the conventional SSD. We can compare it directly to the performance of BwTree_Block, as both are based on the same storage interface. We observe that the conventional SSD is significantly faster than our computational SSD. There are several reasons

for that: (i) the conventional SSD is connected to the host via PCIe instead of fabric, (ii) the FTL of the conventional SSD runs in firmware on a custom ASIC instead of user-space on a SoC running Linux, and (iii) the FTL of the conventional SSD builds on years of experience at Samsung. Our hypothesis is that hardware acceleration will enable us to alleviate the effects of (i) and (ii). Testing this hypothesis is future work.

Interestingly, changing the storage interface from block I/O to batch I/O reduces significantly the gap between conventional SSD and computational storage. With the write-heavy workload under aggressive garbage collection, the performance of `BwTree_Batch` (solid green line in Figure 14(b)) is similar to the performance of the conventional SSD (solid red line), i.e, 2X faster than `BwTree_Block`.

In summary, this experiment shows that an appropriate storage interface compensates for the lack of specialized hardware on computational storage. This is a very encouraging initial result for computational storage.

Our motivation for using a computational SSD was to offload log structuring from host to SSD. Figure 15(a) shows CPU throughput, in operations/sec, as a function of time for the write-heavy workload and a working set that is twice the size of the cache (50% cache). The graph shows that throughput is constant in the non-durable configuration (i.e., in the absence of checkpointing and garbage collection (dotted line)), while it fluctuates at a much lower rate (2/3 of the non-durable configuration) when checkpointing and garbage collection are activated on the host. Figure 15(b) shows that approximately 10% of the CPU load is due to garbage collection. That work is offloaded to computational storage in `BwTree_Batch`.

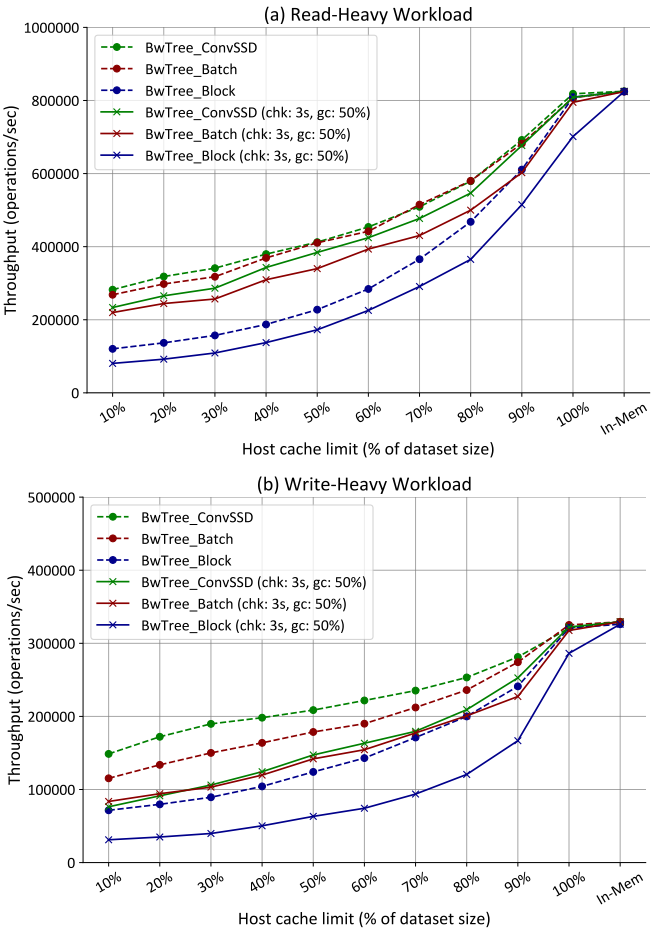

Fig. 14: End-to-end throughput for BW-tree on a conventional SSD compared to `BwTree_Block` and `BwTree_Batch`.

8.3.2 Hardware acceleration

As pointed out in the previous section, we consider that hardware acceleration is necessary to improve the performance of computational storage. In the remaining of this section, we discuss two potential avenues for hardware acceleration: (i) non-volatile memory to reduce the overhead of durability and (ii) hardware-supported RDMA to avoid data copies on computational storage. Integrating these solutions in `OXBatch` is future work.

Non-Volatile Memory If we assume that our programmable SSD contains a form of non-volatile memory (NVM) or DRAM with flash backup in the case of power failure, which enterprise SSDs might do [8], our durability implementation could be significantly simplified. More specifically we would ACK to a request once a buffer arrives in the controller cache. This becomes our latency boundary and should bring the latency down in a few-hundred microsecond or less range. Now pages in the buffer would be written to flash with no worries about the latency of response time, and the entire buffer could be just written to one flash `EBLOCK`. There is no logging nor done record. This approach also reduces thread scheduling and its cost by reducing thread interaction, and also simplifies flash provisioning.

In addition, we need to implement a "checkpoint on power failure" capability. It would be nice to have a power supply that stays on long enough to take the checkpoint involving writing the entire contents in memory to flash storage. We only pay for checkpoints when there is a power failure. The part of flash consumed by the checkpoint is immediately empty and available for reuse when the system comes back up if the checkpoint is written to `EBLOCK`s that are dedicated to the checkpoint.

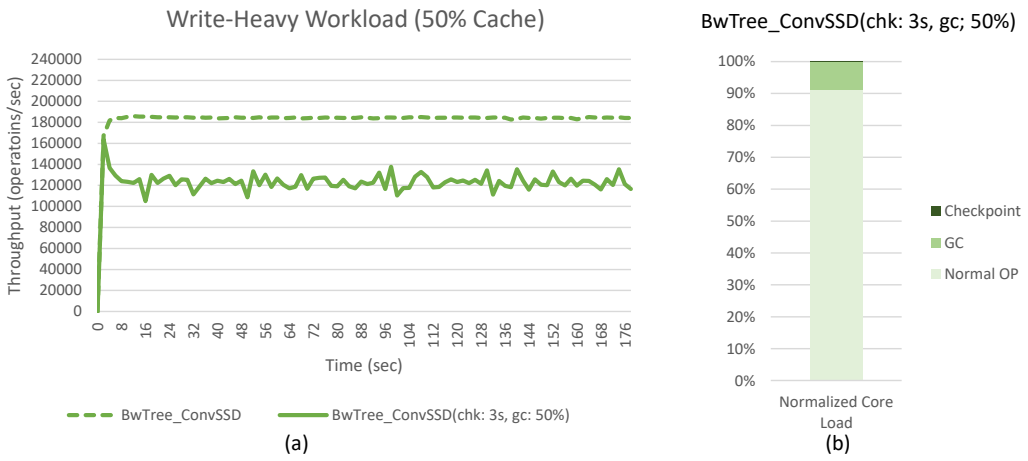

Fig. 15

8.3.3 RDMA

NVMe over Fabrics (NVMe-oF) has been recently introduced as a way for accessing NVMe devices over high-speed networks. With fast interconnect technologies, such as Remote Direct Memory Access (RDMA) or socket connections, NVMe-oF offers a low remote access latency, enabling fast network-based storage I/Os widely used in the industry. As a transport protocol, in our design we used a standard network protocol (i.e., TCP/IP) that offers a cheap and flexible solution in terms of required hardware and software changes to support 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. Such high utilization might be acceptable for a server equipped with a large number of cores so that some can be fully assigned for the network communication purpose while others are used for processing user applications, this is not ideal for storage devices where relatively fewer number of cores exist. To quantify the overhead of processing NVMe-oF (TCP/IP) requests with ARM cores, we measured the performance of transferring 20GB of data with the request size of 64KB over a network interface providing 20Gbps Ethernet bandwidth. We first assigned only a single core to handle the I/O requests, and then increased the number of cores one-by-one.

Figure 16 shows the result. Initially as shown in Figure 16a, we achieved only 500Mbps throughput of transferring the data with a single core dedicated for processing the I/O requests. As increasing the number of cores, higher throughput was observed, but it requires to 100% utilize all the ARM cores (Figure 16b) for fully saturating the network bandwidth. Since all cores are busy only for processing the I/O requests, not much core cycles would be left for running other in-SSD tasks. As a way of improving the interface bottleneck introduced by the socket approach, we could instead use a form of RDMA [21,45] to bypass the CPU whenever possible and to avoid memory copies on the network stack when transferring data. With hardware-support, our hypothesis is that RDMA will significantly improve performance. Testing this hypothesis is a topic for future work.

9 Related Work

9.1 Programmable SSDs

Modern SSDs contain computing components (such as embedded processors and DRAM) to perform various SSD management tasks, providing interesting opportunities to run user-defined programs inside the SSDs. An overview of the concept of programmable SSDs is described in [17]. There is clear industrial interest in exploiting programmable SSDs [2], so research in this area is likely to have a high payoff.

Do et al. [15] were the first to explore such opportunities in the context of database query processing. They modified a commercial database system to push down selection and aggregation operators into a SAS flash SSD. In addition, Jo et al. [26] extended a variation of MySQL to perform early filtering of data by offloading data scanning to an NVMe SSD. While they pioneered the creative use of flash SSDs to open up cost-effective ways of processing data, their approaches were primarily limited by hardware and software aspects of the SSD. Firstly, the embedded processors in the prototype SSDs were clocked at a few hundred MHz and were not powerful enough to run various user-defined programs.

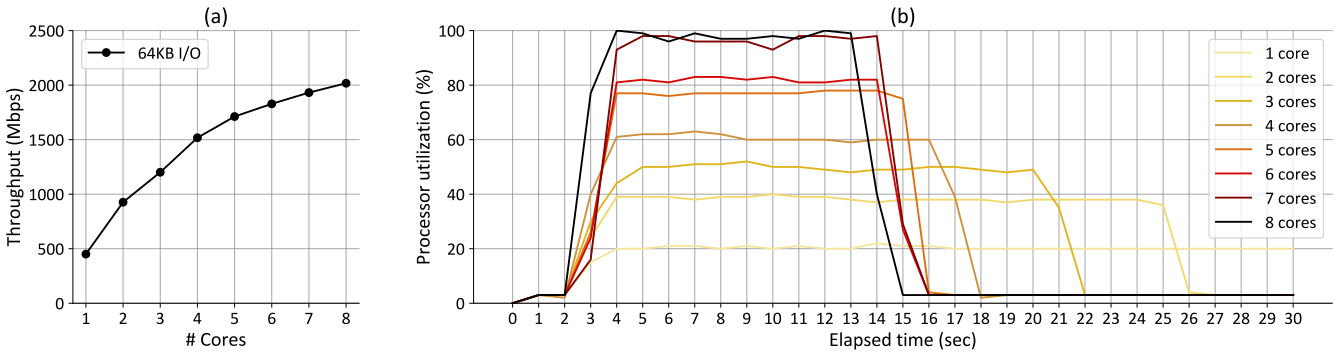

Fig. 16: NVMe over Fabrics (TCP/IP) performance with ARM cores. As the number of cores dedicated to network processing increases, (a) data transfer throughput improves, but (b) more core cycles are required, making fewer cycles to be used for other in-SSD tasks.

More importantly, the software development environments made the development and analysis very challenging, preventing thorough exploration of in-storage processing opportunities.

Recently, researchers have studied better programming models for programmable SSDs. In [40], Seshadri et al. proposed Willow, a PCIe-based generic RPC mechanism, allowing developers to easily augment and extend the SSD semantics with application-specific functions. Gu et al. [20] explored a flow-based programming model where an in-SSD application can be dynamically constructed of tasks and data pipes connecting the tasks. These programming models offer great flexibility of programmability, but are still far from being truly general-purpose. There is a risk that existing large applications might need to be heavily redesigned based on models' capabilities.

9.2 Deuteronomy

The Deuteronomy architecture [31] supports efficient ACID transactions by providing a clean separation of transnational component (TC) from data management component (DC). The idea is to decompose functions of a database storage engine kernel into TC that provides concurrency control and recovery, and DC that handles data storage and management duties (such as access methods, and caching). Each Deuteronomy component is implemented for high performance on modern hardware, resulting in Bw-tree (i.e., a latch-free access method [32]) and LLAMA (i.e., a latch-free, log-structured cache and storage manager [30]). The combination of these two, resulting in a key-value store, is used in several Microsoft products, including SQL Server Hekaton [14] and Azure Cosmos DB [1].

9.3 Our Work

Compared to the earlier studies, our work exploits a state-of-the-art programmable SSD, providing powerful processing capabilities with abundant in-SSD computing resources, and a flexible development environment with a general-purpose operating system which allows easy programming and debugging.

10 Conclusion

Programmable SSDs open up new possibilities for managing the cost/performance trade-off in the storage hierarchy. In this paper, we explored the design of a transactional batched I/O storage interface that removes the burden of Log structuring recovery and garbage collection from the host. We designed and implemented the storage controller software by specializing components from the OX framework. We compared the performance of the Bw-tree key-value store with two storage engines: (i) the LLAMA host-based log structuring running with a traditional block I/O SSD, (ii) a thin layer issuing batched I/O to our storage controller. Our experiments explored the performance implications of offloading log structuring recovery and garbage collection from host to storage controller.

References

1. Microsoft Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/.
2. Microsoft Denali. https://azure.microsoft.com/en-us/blog/microsoft-creates-industry-standards-for-datacenter-hardware-storage-and-security/.
3. NGD Systems. https://www.ngdsystems.com/.
4. NVMe Specifications. https://nvmexpress.org/resources/specifications/.

5. ScaleFlux. https://scaleflux.com/.
6. SNIA Computational Storage. https://www.snia.org/computational/.
7. SPDK FTL. https://spdk.io/doc/ftl.html/.
8. D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong. 2b-ssd: the case for dual, byte-and block-addressable solid-state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 425–438. IEEE, 2018.
9. M. Bjørling, J. González, and P. Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.
10. P. Bonnet. What's up with the storage hierarchy? In *CIDR*, 2017.
11. T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.
12. B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.
13. M. Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.
14. C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton: Sql server's memory-optimized oltp engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.
15. J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230. ACM, 2013.
16. J. Do, D. Lomet, and I. L. Picoli. Improving cpu i/o performance via ssd controller ftl support for batched writes. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, pages 1–8, 2019.
17. J. Do, S. Sengupta, and S. Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6):54–62, 2019.
18. J. González and M. Bjørling. Multi-tenant i/o isolation with open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)*, 2017.
19. J. Gray. Put everything in fugure (disk) controller. *NASD Workshop*, 1998.
20. B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, et al. Biscuit: A framework for near-data processing of big data workloads. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 153–165. IEEE Press, 2016.
21. C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.
22. M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi. The tail at store: A revelation from millions of hours of disk and {SSD} deployments. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 263–276, 2016.
23. X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–9, 2009.
24. J. Huang, A. Badam, L. Caulfield, S. Nath, S. Sengupta, B. Sharma, and M. K. Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 375–390, 2017.
25. Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. Kaml: A flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.
26. I. Jo, D.-H. Bae, A. S. Yoon, J.-U. Kang, S. Cho, D. D. Lee, and J. Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.
27. J. Kim, D. Lee, and S. H. Noh. Towards {SLO} complying ssds through {OPS} isolation. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 183–189, 2015.
28. C. Lee, D. Sim, J. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 273–286, 2015.
29. V. Leis, M. Haubenschild, and T. Neumann. Optimistic lock coupling: A scalable and efficient general-purpose synchronization method. *IEEE Data Eng. Bull.*, 42(1):73–84, 2019.
30. J. Levandoski, D. Lomet, and S. Sengupta. Llama: A cache/storage subsystem for modern hardware. *Proceedings of the VLDB Endowment*, 6(10):877–888, 2013.
31. J. Levandoski, D. Lomet, and K. K. Zhao. Deuteronomy: Transaction support for cloud data. 2011.
32. J. J. Levandoski, D. B. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 302–313. IEEE, 2013.
33. D. Lomet. Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, pages 1–10, 2018.
34. Y. Lu, J. Shu, and W. Zheng. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *Presented as part of the 11th {USENIX} Conference on File and Storage Technologies ({FAST} 13)*, pages 257–270, 2013.
35. P. Mehra. Samsung smartssd: Accelerating data-rich applications. In *Proc. Flash Memory Summit*, 2019.
36. C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.
37. K. Park, Y.-S. Kee, J. M. Patel, J. Do, C. Park, and D. J. Dewitt. Query processing on smart ssds. *IEEE Data Eng. Bull.*, 37(2):19–26, 2014.
38. I. L. Picoli, N. Hedam, P. Tözün, and P. Bonnet. Open-channel ssd (what is it good for). In *CIDR*, 2020.
39. M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.
40. S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable {SSD}. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 67–80, 2014.
41. P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proceedings of the Ninth European Conference on Computer Systems*, page 16. ACM, 2014.

42. J. Xu and S. Swanson. {NOVA}: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 323–338, 2016.
43. J. Zhang, Y. Lu, J. Shu, and X. Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):139, 2017.
44. F. Zhu. Toward the large deployment of open channel ssd. *Flash Memory Summit*, 2019.
45. Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. Congestion control for large-scale rdma deployments. *ACM SIGCOMM Computer Communication Review*, 45(4):523–536, 2015.