Improving CPU I/O Performance via SSD Controller FTL Support for Batched Writes

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

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

Ivan Luiz Picoli
ITU Copenhagen
Copenhagen, Denmark
ivpi@itu.dk

ABSTRACT

Exploiting a storage hierarchy is critical to cost-effective data management. One can achieve great performance when working solely on main memory data. But this comes at a high cost. Systems that use secondary storage as the "home" for data have much lower storage costs as they can not only make the data durable but reduce its storage cost as well. Performance then becomes the challenge, reflected in an increased execution cost. Log structured stores, e.g. Deuteronomy, improve I/O cost/performance by batching writes. However, this incurs the cost of host-based garbage collection and recovery, which duplicates SSD flash translation layer (FTL) functionality. This paper describes the design and implementation in a controller for an Open Channel SSD of a new FTL that supports multi-page I/O without host-based log structuring. This both simplifies the host system and improves performance. The new FTL improves I/O cost/performance with only modest change to the current block at a time, update-in-place interface.

批处理写入

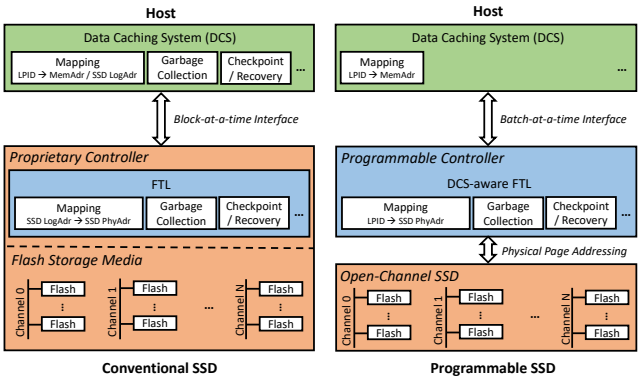

Figure 1: Left: host-based log structuring built using a conventional SSD. Right: host batch I/O interface to log structuring built into an open channel SSD.

1 INTRODUCTION

1.1 Data Management

Increasing main memory size and declining costs have led to more and more data being retained in main memory for extended periods. This reduces the number of I/O's needed to access the data. But I/O is still needed, both to ensure durability for updates, and to reduce storage costs, since secondary storage is substantially cheaper than main memory. The reduction in storage costs comes at the expense of the increased execution cost when an I/O is needed again. This I/O can have a large impact on performance and hence on execution cost.

二级存储的存储成本虽低，但其付出的代价是：I/O的执行成本（数据的访问存取）

In the database domain, main memory systems [6, 19, 33] avoid data access I/O (i.e. bringing data into main memory to operate on it) by retaining all data in main memory. This produces impressive performance. However, systems that cache data temporarily in main memory and selectively move data to and from secondary storage have much lower cost. The technical challenge is to deliver good performance with low cost. Improving I/O performance directly attacks this challenge.

DaMoN'19, July 1, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-6801-8/19/07…$15.00
https://doi.org/10.1145/3329785.3329925

Using a log structured approach [30], e.g. as done in Deuteronomy [21–23], reduces the number of writes to secondary storage by batching large numbers of pages into a single I/O. Unfortunately, this requires implementating log structured re-mapping, garbage collection, and recovery, not an easy task. Further, execution overhead, especially from garbage collection and recovery, negatively impacts performance, reducing the gain from log structuring.

观察图1，FTL虽已具备日志结构所需功能，但DCS不具备

The irony here is that SSD flash drives already support log structuring within their controller for their flash translation layer[FTL] [3], thus duplicating log structuring cpu functionality. Sadly, SSDs support a block-at-a-time interface and the host does not profit from writing blocks in a batch. Open Channel SSDs [1, 29] expect FTL functionality to be provided elsewhere. Thus, we can tailor their controller functionality to provide an FTL with the interface and functionality that we need.

1.2 Main Idea

We describe here a new storage interface. The host system view of storage is of fixed size logical pages (*LPAGE*s) identified via logical page identifiers (*LPID*s). *LPAGE*s to be written are placed into a large buffer (on the order of 256 to 512 blocks in size). I/O is issued for the entire buffer, including a descriptor *DESC* identifying the *LPAGE*s contained in it. Figure 1 illustrates this new interface and how it obviates the need for host-based log structuring.

描述符

The I/O interface resembles the usual storage interface except for batching of a sequence of writes of potentially unrelated "LPAGE"s into a single I/O. Current SSDs, like HDDs, handle only one block at a time. With the new interface, the host needn't maintain a full

mapping table in main memory, as done in LLAMA [21]. Only cached *LPAGE*s need to be mapped to their main memory locations, which is the usual requirement. *LPAGE*s not in main memory do not need host mapping. The host reads an *LPAGE* by providing its *LPID*– indeed that is all it has, since it does not know where the SSD has stored the *LPAGE*.

Batching amortizes the cost of I/O path execution over multiple pages, simplifying and mostly replacing Deuteronomy's LLAMA I/O interface [21] while improving performance. Importantly, the host needn't implement a log structured store. That is done in the Open Channel SSD controller's FTL.

1.3 Paper Organization and Our Contributions

Section 2 describes the software and hardware underlying technology. This includes log structured storage, SSD technology, Open Channel SSDs, and our controller. We implement our log structured capability on this infrastructure, but with a batch write capability instead of a block at a time.

The next sections describe our log structuring technology contributions. These sections describe:

- the multi-page I/O functionality that is the target of our efforts in section 3.
- recovery for the SSD in section 4. Providing a recovery log in the SSD controller presents unique challenges.
- garbage collection within the SSD controller in section 5. Doing this at low cost is challenging.
- checkpointing SSD state in section 6. Frequent checkpoints are needed for fast recovery, requiring efficiency.

We include preliminary performance results in section 7. We discuss related work in section 8. We close with a short discussion in section 9.

2 BACKGROUND

2.1 Flash Solid State Drives (SSDs)

There are two main SSD components [5]: controller and flash storage media.

The controller proprietary firmware, commonly implemented as a system-on-a-chip (SoC), is designed to effectively manage the underlying storage media. SSDs built using NAND flash must be erased before data is written and memory can endure only a limited number of erases before it can no longer be used. To deal with these limitations, the controller implements a form of log structuring [3] that supports a Flash Translation Layer (FTL) to map logical addresses to physical flash addresses and requires garbage collection to reclaim flash blocks containing over-written data, and wear leveling to ensure that flash blocks wear evenly to prolong SSD life.

The storage media is a persistent array of multiple flash channels. A flash channel is for communication between the SSD controller and a subset of flash chips, and a chip consists of multiple blocks, each of which holds multiple pages. The unit of erasure is a block while the read and write operations are done at the granularity of pages. To obtain higher I/O performance from the flash storage media, channel and chip level interleaving techniques are usually employed.

Table 1: SSD Terms

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

2.1.1 In-SSD processing: Modern SSDs package processing and storage components for routine tasks. These computing resources present an opportunity to run user-defined programs inside the SSD [7, 16, 28], with several in-SSD programming models [12, 32] providing great functional flexibility. For example, emerging commercial [31, 34] and research [2, 18] SSD controllers include general-purpose, multi-GHZ clock speed, multi-core processors with built-in hardware accelerators 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.

Programming SSDs is abetted by the trend away from proprietary architectures and software runtimes and toward commodity operating systems running on general-purpose processors. This enables developers to leverage existing tools, libraries and expertise rather than spending long hours learning a low-level, embedded development process. This also allows easy porting of existing software running on host operating systems to the device.

2.1.2 Exposing SSD internals: The widespread adoption of SSDs in data centers was facilitated by their support of the same block-at-a-time interface as traditional hard disk drives (HDDs). But this led to sub-optimal storage utilization and performance [13, 17, 25]. To address this requires exposing SSD's internal media geometry and parallelism so that a host can control data placement and I/O scheduling in a better way based on user requirements [10, 14]. Parts of the industry are moving in this direction: Open-Channel SSDs [1, 29], for example, introduces a new I/O interface that enables the host to access physical flash pages, not possible with the traditional block device abstraction.

We will frequently refer to hardware elements of SSDs. Table 1 provides the terms we will use subsequently.

2.2 Log Structuring

Log-structured file systems (LFS) [30] were introduced to leverage high sequential performance of hard disks. LFS batches random updates in a large main memory buffer first, and then sequentially writes the whole buffer as a large segment to the HDD. SSDs also favor sequential writes over random writes to achieve high write performance [8], so log structured techniques have been naturally explored as a way for a host to exploit SSDs [20, 37].

Although using a host-based log structured approach reduces the number of I/O operations, efficiently implementing the approach is complex because disk space occupied by stale data has to be reclaimed to ensure contiguous free areas for sequential writes. SSDs already use log structuring approaches in their FTLs to overcome the limitations of flash memory such as no in-place updates. Both host LSF and SSD FTL implement their own mapping, cleaning, and recovery mechanisms, and these redundant functionalities result

	HOST	SSD
Host-based LS	LPID ➡ LogAdr	LogAdr ➡ SSDAdr
SSD-based LS	LPID	LPID ➡ SSDAdr

Figure 2: The double mapping of host logical pages with their *LPID*s to SSD physical storage locations. Host-based LS and SSD-based LS denote host-based log structuring and our SSD controller based log structuring, respectively.

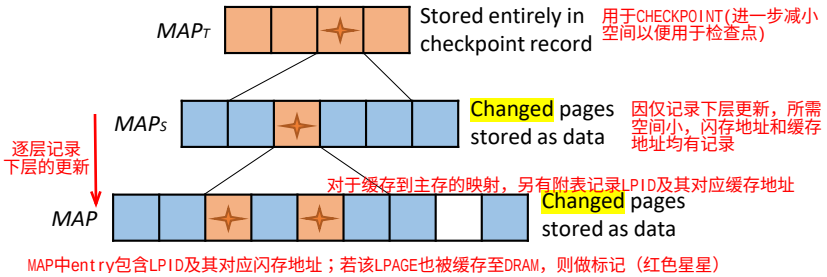

Figure 3: The hierarchical structure used in managing the cached part of *MAP* and producing an incremental checkpoint for it.

in unnecessary management overhead. Recent research has targetted this overhead for elimination or reduction so as to optimize log structured key-value stores using SSDs with the open-channel architecture [15, 36, 38].

3 MULTI-PAGE WRITES

3.1 Batching I/Os

Log structuring replaces multiple write I/Os, one for each *LPAGE* (block), with a single I/O of a large buffer containing the *LPAGE*s. Batching the pages enables sharing of a large part of the I/O path to the secondary storage device. As discussed in [24], this impacts both performance and cost of operations requiring an I/O.

*LPAGE*s within a log structured buffer are time ordered, i.e., the updates to pages are intended to be in the order of the *LPAGE*s within a buffer, and between buffers. This order determines the visible states of updated blocks. Reads must expose SSD state as reflected in the write order:

- a read preceding a write batch (Ack'd before the write batch is initiated) sees no updates of the batch.
- a read following a write batch (issued after the write batch is Ack'd) sees all updates in the batch.
- a read concurrent with a write batch (initiated and/or Ack'd between the times a write batch is issued and Ack'd) sees a state that includes a time ordered prefix of the blocks in the batch.
- a read B concurrent with a write batch but later than an earlier Ack'd read A sees a prefix of the blocks of the batch that includes the prefix seen by A.
- either all writes of a batch eventually appear in the data state on the SSD or none of them do.

读取的确认在写批处理的发起之前
读取的发起在写批处理的确认之后
读取的发起或确认在写批处理的发起和确认之间
时间顺序前缀

3.2 Log Structuring Functionality

3.2.1 Host Log Structuring. Host-based log structuring requires the data management system, e.g., Deuteronomy, to assign a "physical address" for each logical page (*LPAGE*), identified with a logical page id (*LPID*). The host translates *LPID*s into SSD addresses when accessing secondary storage. 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 its logical locations, and maps them to its own private physical locations, which are unknown to the host. These mappings are illustrated in Figure 2.

Thus, the SSD, when garbage collecting old versions, simply trims large blocks. The host must do finer grained *LPAGE* level garbage collection. This requires the host to read its previously written buffers back into main memory to relocate still valid *LPAGE*s.

即后文说的：host based SSD address

The host-based storage management must provide durabililty. Not only must the data in an *LPAGE* be made durable. The host needs also to durably maintain the mapping between *LPID*s and physical locations up to the last acknowledged write. Deuteronomy uses redo recovery coupled with periodic checkpointing to ensure this durability.

3.2.2 SSD-Controller Log Structuring. Our SSD controller based log structuring accepts large, multi-page buffers. Unlike the host based system, where the SSD knows nothing about the logical blocks within a buffer, the *LPAGE*s 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, i.e. an ordered set of <*LPID*, offset> pairs (offsets can be computed when we use fixed size *LPAGE*s), the ordering reflecting the write order for the blocks in the buffer.

The SSD controller maintains a table (*MAP*) mapping *LPAGE*s to the physical SSD locations where they are written (see Figure 2). [In SSDs, this is usually referred to as the flash translation layer or FTL.] The SSD is now responsible for:

- 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 since *MAP* (which is the major recoverable element in the system) is under its management.

MAP is organized as a hierarchical search structure (see Figure 3) that we exploit in cache management and incremental checkpoints. It is indexed by *LPID*, and each entry contains an 8 byte flash address to the durable location of an *LPAGE*. Because of caching, some *LPAGE*s may also be cached in SSD DRAM. Thus, a *MAP* entry also contains a flag for this. An associative side table (much smaller than *MAP*) is used to locate cached *LPAGE*s. This avoids potential doubling of *MAP* were it to always include both flash and DRAM addresses.

MAP's size is 4GB for our 2TB flash drive and is not entirely in controller DRAM cache. It is paginated so that large parts of it do not consume DRAM. Pages of *MAP* are found via a smaller table MAP_S. MAP_S tracts *MAP* pages in the same way that *MAP* tracks *LPAGE*s. Each entry of MAP_S includes not only a flash address, but also a cache address because (1) MAP_S is much smaller than *MAP*, and (2) *MAP* pages are likely to be cached in DRAM.

标有页数

The complete MAP_S table with 16 byte entries has a size of 16MB and is entirely in contiguous locations of controller DRAM memory. An *LPAGE* is found by: (1) using a 20 bit *LPID* prefix as an index to the MAP_S entry. This entry 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 *MAP* pages needed for active *LPAGE*s.

4 DURABILITY AND RECOVERY

4.1 Log Structuring Crash Recovery

The SSD's log structured store must ensure the durability of *MAP* and *LPAGE* states up to and including the last acknowledged buffer write prior to the system crash. This recovery is not needed for conventional update-in-place storage systems.

MAP can be thought of as part of the SSD "database". Every update for an *LPID* also updates the "*MAP* database" with a new physical flash address for its *MAP* entry. While *LPAGE*s written by the host are the source for updates, the recoverable updates are the changes to *MAP* entries.

Of course, the SSD itself provides durability. Whatever is written to a flash *WBLOCK* (see Table 1) is durable. Should the write fail, *WBLOCK* state can be determined by attempting to read it. *RBLOCK*s each have a *TAG* field where we store the *LPID* associated with the *RBLOCK* and a log sequence number for the update. This makes recovery possible by reading *TAG*s across the entire SSD. But we want much faster recovery. So instead, we use database style recovery.

Database recovery exploits write-ahead logging (WAL). WAL ensures that updates are stable on the log ahead of them being entered into the database. If a failure occurs, the log is replayed from a checkpointed state to recover current database state. The length of the log is controlled by periodic checkpoints, which enable recovery time to likewise be controlled. Thus, should an SSD controller crash, we replay the SSD "log" to recover *MAP*, using *LPAGE* physical locations as the updated values for *MAP LPID* entries.

4.2 Normal Operation

4.2.1 Update Protocol. Our recovery log is derived from the *DESC* blocks that accompany the buffer of blocks written in each batch. A *DESC* associates *LPID*s with buffer blocks. The SSD associates an *LPID* with the physical flash address at which we store its data. This is illustrated in Figure 4. The mapping metadata in *DESC* is used to create an (*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 this log write to be ack'd before we update the associated SSD *LPAGE* physical locations.

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

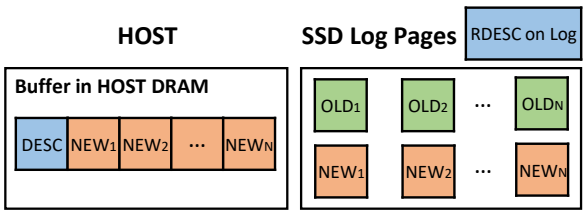

Figure 4: Transformation of *DESC* describing blocks in a write buffer identified by *LPID*s to a redo log recovery block *RDESC* that describes where the data for each *LPID* has been written on flash

We write *LPAGE*s to their *RDESC*-designated flash addresses in parallel using multiple threads and flash channels. This is the largest step in making *LPAGE*s durable and we exploit parallelism for low latency. We use a single thread to update *MAP* in *RDESC* order to guarantee the semantics of section 3. Crash recovery uses the same technique to update *MAP*.

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 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 *LPID*s 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 $RDESC_A$s until we have successfully written all blocks. Updating *MAP* does not start until all blocks are written successfully to flash. Once *MAP* is updated, an ACK is returned to the host system.

How does recovery know, after the loss of *MAP* in controller volatile state, whether a set of *LPAGE* from an *RDESC* have been updated since we start on the next *RDESC* before we have finished writing the updated pages of prior *RDESC*s? Verifying each write in recovery log *RDESC* is expensive, requiring reading each physical address to determine whether the write succeeded. To avoid this, we write a **done** record for an *RDESC* when its updates are completely installed. This does not require a separate log write, merely a notation for it in a subsequent *RDESC* log write. We need not wait for the **done** record to be durable before ack'ing the writing of *RDESC* data blocks.

4.3 Linked List Recovery Log

Recovery logs are usually physically sequential. But we want to make *RDESC*s for the log durable promptly since writing *RDESC*s is in the latency path of a host I/O operation. Further, SSD storage is partitioned, consumed and re-cycled in relatively divergent ways. So we use a doubly linked list for our logically (not physically) sequential log. Each *RDESC* has a back link that points back to its predecessor block, and a FORWARD link that points to the location of the successor block. This permits the recovery log to be traversed both forward and backwards without being physically sequential.

Log writes can 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. We tolerate as many log write failures as there are forward links in the vector.

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 makes our log an undo/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.

5 SSD GARBAGE COLLECTION

5.1 Erase Block Based Garbage Collection

Flash storage does not support update in place, but needs an erase operation on an erase block (*EBLOCK*) of 8MB before re-writing it anew. This is why SSDs use log structuring for their FTLs. As a result of log structuring, old versions "hang around" until garbage collected.

To exploit SSD controller parallelism, pages of the host write buffer are written across multiple *EBLOCK*s, in write block (*WBLOCK*) units. However, an *EBLOCK*, being the erasure unit, is the unit of garbage collection. Thus, we need to determine, per *EBLOCK*, the *RBLOCK*s that have been over-written by subsequent *LPage* updates. We need garbage collection (GC) to reclaim their storage. Each still current *RBLOCK* of an *EBLOCK* **old** is re-written to a new location in another *EBLOCK* **new** that was previously erased, and is thus writable. Once all *RBLOCK*s in **old** have been "overwritten", **old** can be erased and re-used.

5.2 GC, Bit Vectors, and TAGs

We need metadata associated with an *EBLOCK* , outside of the physical *RBLOCK*s to which logical pages are mapped, to track

区分失效数据页和有效数据页

- which *RBLOCK*S are garbage and which are not;
- how many *RBLOCK*s are garbage in an *EBLOCK*; 记录失效数据页数量
- which logical pages that are not garbage need their *RBLOCK*s to be relocated to another *EBLOCK* and their MAP entry updated. 记录有效数据页的迁移信息

We maintain a global bit vector GBITV to indentify which *RBLOCK*s are garbage and which are not. The segment of GBITV associated with an *EBLOCK* also tells us how much *EBLOCK* storage is garbage. We schedule *EBLOCK*S for garbage collection in available storage (amount of garbage) order. Reclaiming this storage is the payoff for GC.

We use an *EBLOCK*'s GBITV segment to identify still valid pages (*RBLOCK*s) in an *EBLOCK*. GC reads each still valid *RBLOCK* and its associated *TAG* field. The *TAG* tells us the *LPID* for the *RBLOCK MAP* entry. GC 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 DRAM.

We have implemented a very simple scheme for separating hot data from cold data, a technique found to improve the GC efficiency of log structured file systems [30] (i.e., reducing the *RBLOCK*s needing to be re-written per ~~*RBLOCK*~~ of storage reclaimed). This reduces the re-writing of cold data, and focuses most GC on the current (presumed hot) data. GC uses a different set of *EBLOCK*s for re-writing *RBLOCK*s of *EBLOCK*s being cleaned than is used for ordinary user updates. There are other more sophisticated strategies that can improve garbage collection efficiency further [30].

5.3 Protecting the Recovery Log

Section 4.3 described the recovery log as a linked list of *RDESC*s. But *RDESC*s 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*, 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 a log page. We have no simple way of relocating log pages while maintaining the log's linked list. Instead, we mark an *EBLOCK* as NOT subject to garbage collection until a checkpoint truncates the part of the recovery log that is in the *EBLOCK*.

6 SSD CHECKPOINTING

Checkpoints are used to truncate 截短 the part of the recovery log used to restore SSD state after a crash. Should our SSD controller crash, we apply redo log records that follow the latest checkpoint to this latest completed checkpoint. End of log is detected when the next pointers in the recovery log point only to erased blocks.

6.1 Fuzzy Checkpointing in Recovery

We use a fuzzy 失真 incremental checkpoint to truncate the recovery log. It is an application of techniques used in database systems, e.g. [27]. A checkpoint is fuzzy when it captures the state over some time interval, and uses redo recovery to make the state precise as of the end of the interval. A checkpoint is incremental when it only captures the part of the state that has changed since a prior checkpoint and merges that with the unchanged part. We show the elements of the state that we capture for a checkpoint in Table 2.

We introduced hierarchies to facilitate saving the states of *MAP*, *GBITV*, and *WEAR* incrementally. We save these states by treating their pages, other than the root of the hierarchy, 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 (mapping 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 posting all changes to this metadata to the SSD, we complete the checkpoint by writing the checkpoint record in a "well-known location".

6.2 Mapping Information

MAP and related mapping information is the largest element of our state. *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. Such an update marks its *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

Table 2: Recoverable State Terms

MAP	Mapping Table: *LPID* - physical flash address
MAP_S	Small table paginating *MAP*
MAP_T	Tiny table paginating MAP_S
GBITV	Bit vector: *RBLOCK*s 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*

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) table because we do not want to write all of MAP_S 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. It uses the first 12 bits of an *LPID* to identify a page in MAP_S (in the same way that MAP_S identifies a page in *MAP*). When we persist MAP_S pages that have changed, these update entries in MAP_T. MAP_T is not on the access path to data blocks. Accesses all start at MAP_S.

The final checkpoint step is to include MAP_T in its entirety as part of the checkpoint state in a "well-known" location.

6.3 Other State

6.3.1 Garbage State. Garbage is tracked using a bit vector per flash block, each bit representing the state of a 4KB page. The cumulative size of the bit vectors (called *GBITV*) is 64MB, 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 128KB. $GBITV_S$'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 bytes, as we did MAP_T before for mapping data. $GBITV_T$ is stored in its entirety in the checkpoint record.

6.3.2 Wear State. Each flash block (the unit of erasure) of 8MB has a limited number of erase cycles. We count and maintain a 4 byte counter per block 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 store directly into our checkpoint record, without a tiny wear state vector.

6.4 Checkpoint Record

The complete state that we need to maintain across system crashes, after having converted larger tables to updatable data, then consists of MAP_T (16KB), $GBITV_T$ (256 bytes), $WEAR_S$ (2KB), and the log location of where to start recovery using the recovery log (8 bytes), called the "redo scan start point" or RSSP. This comprises our checkpoint record that we write to a "well known location" so that it can be found after a crash.

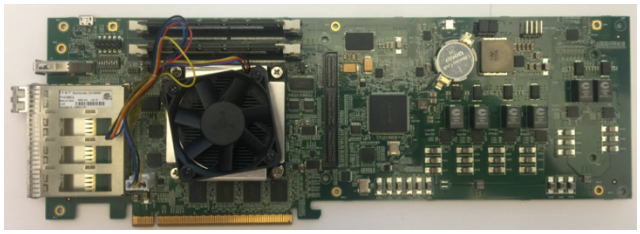

Figure 5: DragonFire Card (DFC)

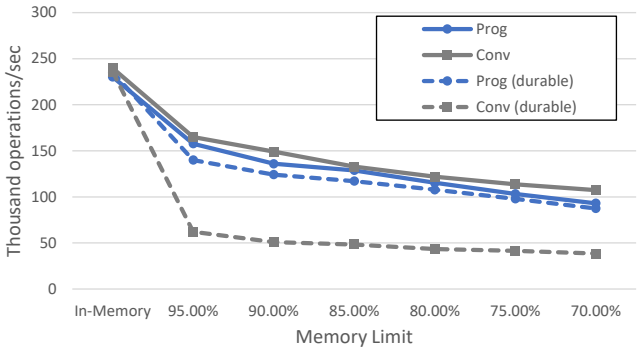

Figure 6: YCSB throughput

7 EXPERIMENTS

7.1 Experimental Setup

We ran all the experiments on a single machine with an Intel Xeon Silver 4109T Core @ 2 GHz and 32 GB of DRAM, with two different types of SSDs. For our baseline conventional SSD, we used a 512GB NVMe/PCIe SSD, rated at around 1.2 GB/sec for random 4KB reads and around 2 GB/sec for sequential writes. Our programmable SSD is built based on the DragonFire Card (DFC) [2] (shown in Figure 5) designed by DellEMC/NXP equipped with an ARM Cortex-A72 processor. This SSD internally uses a CNEX open-channel SSD [1] for issuing read, write and erase operations against raw flash memory, rated at around 1.6 GB/sec for random 4KB reads and 2.4 GB/sec for sequential writes. We used a single cpu core so that our experiments are never I/O bound. We run Ubuntu Linux 16.04 LTS on both the host and the programmable SSD,

We used a key-value store based on Bw-Tree [21, 22], a latch-free, B-tree style index layered on LLAMA to compare the performance of log structuring built using conventional and programmable SSDs. Our benchmark is YCSB [4] with a read-mostly workload (10 million operations of 75% reads and 25% updates with an uniform distribution). YCSB is a widely used framework for evaluating performance of NoSQL stores. Our YCSB database had 5 million records, each record with a 8-byte key, and 100-byte 10 fields. We set the Bw-Tree page size to 4 KB. When enabled, we triggered a checkpoint every 10 seconds and targeted garbage collection (GC) for a 100% space overhead. We experimentally set the I/O queue depth size to be large enough that the queue is never starved.

7.2 Preliminary Results

We present the overall throughput achieved for YCSB when the Bw-tree key-value store runs with (1) conventional (*Conv*) and (2) programmable (*Prog*) SSDs, using a single host core. We varied the host DRAM cache size over a range measured as a percentage of database size (Figure 6). For each SSD configuration, we measured performance with both GC and checkpointing enabled (durable) and with those disabled (non-durable).

As shown in the figure, in the non-durable experiments, *Conv* had slightly better performance than *Prog*. When a non-cached page is requested, the Bw-tree needs to first find the SSD location where the page is stored, which is performed on the host (*Conv*) or in the SSD (*Prog*). Unlike write I/Os, our Bw-tree always reads one page at a time, and therefore the weaker processor used in *Prog* introduces a longer read latency than *Conv*. On the other hand, *Prog (durable)* is between 1.91x - 2.49x faster than *Conv (durable)* in the "durable" configuration. In this configuration, the throughput of *Conv (durable)* is significantly restricted by its need to run checkpoint and GC on the same host core. In contrast, those processes for *Prog (durable)* are offloaded to the SSD controller, enabling host cycles to be dedicated to handling benchmark operations.

8 RELATED WORK

8.1 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 [9]. There is clear industrial interest in exploiting programmable SSDs [35], so research in this area is likely to have a high payoff.

Do et al. [7] 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. [16] 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. 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 [32], 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. [12] 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.

8.2 Deuteronomy

The Deuteronomy architecture [23] 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 [22]) and LLAMA (i.e., a latch-free, log-structured cache and storage manager [21]). The combination of these two, resulting in a key-value store, is used in several Microsoft products, including SQL Server Hekaton [6] and Azure Cosmos DB [26].

8.3 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.

9 CONCLUSION

Our focus has been to use the new SSD controller capabilities to attack the I/O problem, first highlighted by Gray et al's 5 minute rule [11]. Our DaMoN paper [24] demonstrated the unpleasant performance impact resulting from the high cost of executing the I/O path when providing data management in a data caching system.

Efficiency of moving data between layers of a storage hierarchy is the key to enabling high performance at low cost in data stores. Cold data needs low cost storage, while hot data needs high performance storage. Moving between storage layers can manage data stores for best cost/performance, while non-caching stores all too frequently sacrifice either cost or performance.

We have used programmable SSD controllers to implement a storage interface that exploits batching of page I/O to enable the I/O cost to be amortized across multiple pages. This is similar to log structuring techniques. But by placing this functionality into the storage controller, we have enabled log structuring overhead to be removed from the host system. This both removes redundant functionality and improves the cost performance of the resulting I/O interface.

REFERENCES

[1] M. Bjørling, J. González, and P. Bonnet. Lightnvm: The linux open-channel ssd subsystem. In *FAST*, pages 359–374, 2017.

[2] P. Bonnet. What's up with the storage hierarchy? In *CIDR*, 2017.

[3] 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.

[4] B. Cooper, A. Silberstein, E. Tam, R. Ramakrrishnan and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[5] M. Cornwell. Anatomy of a solid-state drive. *Commun. ACM*, 55(12):59–63, 2012.

[6] 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. ACM, 2013.

[7] 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.

[8] J. Do and J. M. Patel. Join processing for flash ssds: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, pages 1–8. ACM, 2009.

[9] J. Do, S. Sengupta, and S. Swanson Programmable Solid-State Storage in Future Cloud Datacenters. Communications of the ACM, 2019.

[10] J. González and M. Bjørling. Multi-tenant i/o isolation with open-channel ssds. In *Nonvolatile Memory Workshop (NVMW)*, 2017.

[11] Jim Gray, Gianfranco R. Putzolu: The 5 Minute Rule for Trading Memory for Disk Accesses and The 10 Byte Rule for Trading Memory for CPU Time. SIGMOD Conference 1987: 395-398

[12] 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.

[13] M. Hao, G. Soundararajan, D. R. 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 *FAST*, pages 263–276, 2016.

[14] 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 *FAST*, pages 375–390, 2017.

[15] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. Kaml: A flexible, high-performance key-value ssd. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*, pages 373–384. IEEE, 2017.

[16] 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.

[17] J. Kim, D. Lee, and S. H. Noh. Towards slo complying ssds through ops isolation. In *FAST*, pages 183–189, 2015.

[18] G. Koo, K. K. Matam, H. Narra, J. Li, H.-W. Tseng, S. Swanson, M. Annavaram, et al. Summarizer: trading communication with computing near storage. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 219–231. ACM, 2017.

[19] J. Lee, M. Muehle, N. May, F. Faerber, V. Sikka, H. Plattner, J. Krueger, and M. Grund. High-performance transaction processing in sap hana. *IEEE Data Eng. Bull.*, 36(2):28–33, 2013.

[20] C. Lee, D. Sim, J. Y. Hwang, and S. Cho. F2fs: A new file system for flash storage. In *FAST*, pages 273–286, 2015.

[21] 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.

[22] J. Levandoski, D. Lomet, and S. Sengupta. The bw-tree: A b-tree for new hardware platforms. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 302–313. IEEE, 2013.

[23] J. Levandoski, D. Lomet, M. Mokbel, and K. Zhao. Deuteronomy: Transaction Support for Cloud Data. In *CIDR*, 2011.

[24] 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*, page 9. ACM, 2018.

[25] Y. Lu, J. Shu, W. Zheng, et al. Extending the lifetime of flash-based storage through reducing write amplification from file systems. In *FAST*, volume 13, 2013.

[26] Microsoft Azure Cosmos DB. https://azure.microsoft.com/en-us/services/cosmos-db/

[27] 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.

[28] 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.

[29] I. L. Picoli, C. V. Pasco, B. Þ. Jónsson, L. Bouganim, and P. Bonnet. uflip-oc: Understanding flash i/o patterns on open-channel solid-state drives. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, page 20. ACM, 2017.

[30] 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.

[31] ScaleFlux. Computational storage: Acceleration through intelligence & agility. *Flash Memory Summit*, 2018.

[32] S. Seshadri, M. Gahagan, M. S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, 2014.

[33] M. Stonebraker and A. Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[34] N. Systems. Intelligent storage produces efficient scalable system. *Flash Memory Summit*, 2018.

[35] K. Vaid. Microsoft creates industry standards for datacenter hardware storage and security. *https://azure.microsoft.com/en-us/blog/microsoft-creates-industry-standards-for-datacenter-hardware-storage-and-security/*, 2018.

[36] 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.

[37] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *FAST*, pages 323–338, 2016.

[38] 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.