2025 Design, Automation & Test in Europe Conference (DATE 2025)

Ensuring Data Freshness for In-Storage Computing with Cooperative Buffer Manager

Jin Xue Yuhong Song Yang Guo Zili Shao
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China
Email: {jinxue, yhsong, yangguo, shao}@cse.cuhk.edu.hk

***Abstract*—In-storage computing (ISC) aims to mitigate the excessive data movement between the host memory and storage by offloading computation to storage devices for in-situ execution. However, ensuring data freshness remains a key challenge for practical ISC. For performance considerations, many data processing systems implement a buffer manager to cache part of the on-disk data in the host memory. While the host applications commit updates to the in-memory cached copies of the data, ISC operators offloaded to the device only have access to the on-disk persistent data. Thus, ISC may miss the most recent updates from the host and produce incorrect results after reading the stale and inconsistent data from the persistent storage. With this limitation, current ISC can only be used in read-only settings where the on-disk data are not subject to concurrent updates. To tackle this problem, we propose a cooperative buffer manager for ISC to transparently provide data freshness guarantees to host applications. Proposed methods allow the device to synchronize with the host buffer manager and decide whether to read the most recent copy of data from host memory or flash memory. We implement our method based on a real hardware platform and perform evaluation with a B+-tree based key-value store. Experiments show that our method can provide transparent data freshness for host applications with reduced latency.**

***Index Terms*—in-storage computing, data freshness, solid-state drives**

I. Introduction

Modern storage technologies such as flash memory have greatly reduced storage medium access time which once dominated the overall performance of large-scale data processing systems. However, despite the advancements of storage devices, traditional system architectures still treat them as passive data storage, *i.e.*, data are brought into the host memory for processing and then in-memory results are written back, causing a high data movement overhead between the host and device. As a result, the system bottleneck shifts from the storage access latencies to the host-device interconnect bandwidth and host I/O stack overhead [1], [2], [3].

To reduce the data movement overhead, one potential solution is in-storage computing (ISC) [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. With ISC, computation can be pushed down to storage devices, such as solid-state drives (SSDs), and executed with in-storage processing elements. ISC is particularly suitable for tasks such as data filtering and aggregation as the volume of output data that need to be transferred to the host is greatly reduced.

The work described in this paper is partially supported by the grants from the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 14219422, GRF 14202123).

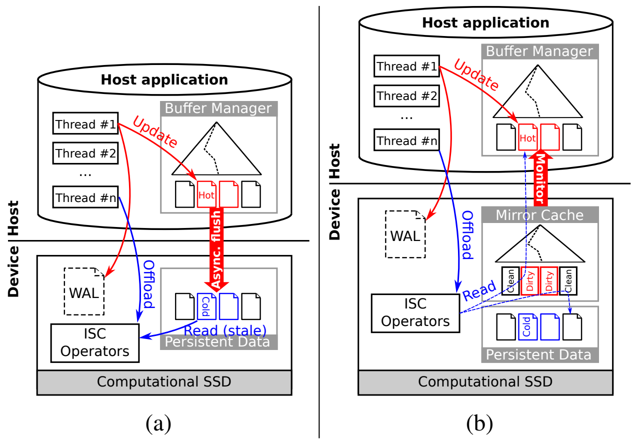


Fig. 1: Data freshness problem of in-storage computing. (a) In existing applications with a host buffer manager, ISC operators offloaded by the host may miss the most recent updates that are cached in the host memory and read stale data from the flash memory. (b) With the proposed cooperative buffer manager, the device keeps track of the state of the host buffer manager so that ISC operators can decide where to retrieve data based on whether the host buffer manager contains a dirty copy.

Moreover, by offloading compute tasks to the storage devices, host systems can utilize more processing power on other tasks.

However, one key challenge brought forward by in-storage computing is **ensuring data freshness** between the host and device. For performance considerations, many data processing systems implement a buffer manager to cache part of the on-disk data as fixed-size pages (usually 4-64KB) in the host memory, as shown in Figure 1(a). A typical example of such applications is modern database management systems (DBMSs). When an update thread needs to modify a disk page, it acquires a cached copy from the buffer manager and applies the updates directly on the cached page. The modified pages are only written back to the stable storage later when they are evicted by the buffer manager or during database checkpoints. To ensure data integrity, an update thread also needs to write a log record describing all its modifications to the write-ahead log (WAL) on the storage device before it commits. Meanwhile, other threads may submit offloaded ISC operators to the device for execution, which will miss the most recent updates and perform computation on the stale data read from the persistent storage, breaking strong consistency guarantees. Pushing computation to the device may not be feasible, if there is a more current copy of the data in the host buffer pool than that on the SSD [2]. Due to this limitation, existing systems only utilize in-storage computing in read-only settings [3], [5], [6], [8], [10], [12], where the on-disk data are not subject to concurrent updates.

In this paper, we propose a cooperative buffer manager (CBM) for in-storage computing to transparently provide *data freshness guarantee* to host applications that manage their own page buffers with hybrid logging. As shown in Figure 1(b), the computational storage device maintains a mirrored version of the host buffer manager in the device controller memory, which maps pages to their *cache state* (*e.g.*, whether they have been modified at the host side). ISC operators offloaded to the storage device can thus determine if the host buffer manager contains hot data for the page, and retrieve the freshest data without host intervention.

The contributions of this paper are summarized as follows:

- We propose a cooperative buffer manager (CBM) that utilizes a device-side mirror cache to keep track of host-side cache state and provide transparent data freshness to ISC-enabled applications without host intervention.
- Meanwhile, a page state piggybacking mechanism is proposed that allows the host to propagate updates of the cache state to the device *along with* existing I/O commands, which avoids extra synchronization.
- We implement CBM on a real hardware platform, and evaluate the performance with real application workloads on a B+-tree-based key-value store, which shows that our methods can provide transparent data freshness with reduced latency and less data transfers.

II. Background and Related Works

A. Data Freshness for In-Storage Computing

ISC has emerged as a promising solution to the excessive data movement between host systems and storage devices. As modern SSDs equipped with high-end multi-core microprocessors and other computational resources become more common, it is feasible to offload complex compute tasks to execute on storage devices. With ISC, computational tasks such as data filtering and compression can significantly reduce the size of output data and thus the amount of data transferred to the host.

There are several works [17], [18] bringing compute capabilities to smart SSDs to enable efficient in-storage computing. However, these systems either utilize ISC under read-only settings, such as performing analytical requests on static data that are not subject to frequent changes, or require all writes to be mediated by ISC operators. In reality, data may be read and written by the host and device independently at the same time. Even worse, host applications may cache part of on-disk data in the host memory, creating two inconsistent copies of the same data. If the updates made by the host cannot be propagated to the device promptly, ISC operators may read stale and inconsistent data from the persistent storage and produce incorrect results. How to propagate modifications promptly from the host to the storage device to ensure data consistency and freshness remains an open question for practical ISC.

TABLE I: Qualitative comparison of ISC systems that provide data freshness guarantee. We consider the following characteristics: (1) Flush size: the amount of data that needs to be flushed to the device to guarantee data freshness. (2) Code modifications: if host applications need to be modified to explicitly propagate updates. (3) Access pattern: what kinds of access patterns are supported for the ISC applications.

System	Flush size	Code modifications	Access pattern
INSIDER [3] and λ-**IO** [5]	Large	No	Sequential-only (Streaming)
Update-aware NDP [19]	Medium	Application-specific	Sequential/ Random
Ours	Small	No	Sequential/ Random

B. Related Works

To ensure data consistency, INSIDER [3] and λ-IO [5] extend the vanilla POSIX file interface and allow user applications to submit offloaded computation during file data transfer. These systems leverage the existing components provided by the host operating system (OS) kernel, such as virtual filesystem (VFS) and page cache, to let the host control data consistency. However, INSIDER and λ-IO assume a streaming processing pattern in which the ISC operators scan through all data within the specified file range, and thus the host system can determine whether a dirty page needs to be flushed to the device by testing if it falls within the requested range. For random data access patterns, it is a non-trivial task for the database to determine which updates are required for an ISC operator and need to be propagated, while blindly flushing all dirty pages to the device can cause a large amount of data to be transferred, leading to high write amplification.

Alternatively, update-aware NDP [19] provides data freshness guarantee to offloaded read-only analytical (OLAP) operations in the presence of concurrent update transactions in a hybrid transactional and analytical processing (HTAP) database. In the update-aware NDP architecture, all incoming modifications to the database are accumulated in shadow database-specific data structures collectively referred to as the *shared state*. The shared state is either flushed periodically or eagerly passed along to the device before dispatching ISC requests. However, update-aware NDP requires application-specific code modification to manage the shared state, which introduces extra manual overhead with low flexibility.

We summarize various characteristics of the related ISC systems through a qualitative comparison in Table I. Our method aims to provide transparent data freshness to ISC-enabled applications, which only flushes a small portion of updated data on-demand, without requiring any code modifications, and supports both sequential and random access patterns.

III. Proposed Cooperative Buffer Manager

Indeed, the crux of the data freshness problem of ISC is to identify all dirty states scattered across the host address

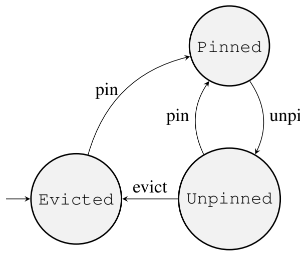
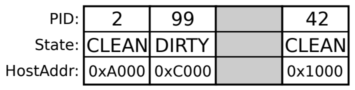


Fig. 2: State transitions of data pages in the host buffer manager.



PID:	2	99		42
State:	CLEAN	DIRTY		CLEAN
HostAddr:	0xA000	0xC000		0x1000

Fig. 3: Structure of the on-device mirror cache.

Algorithm 1: Read page function for ISC operators.

```
Data: Request page ID $pid$, device DRAM buffer to
      hold page data $pageBuf$
function ReadPageISC($pid$, $pageBuf$):
    $desc \leftarrow$ LookupPageDesc($pid$);
    if $desc == \varnothing$ or $desc.state ==$ Clean then
        /* Evicted or clean page          */
        $offset \leftarrow pid * pageSize$;
        ReadFromFTL($pageBuf, pageSize, offset$);
    else
        /* Dirty page                     */
        repeat
            ReadFromHost($pageBuf, pageSize,$
              $desc.hostAddr$);
            $fpv \leftarrow$ GetFrontPageVer($pageBuf$);
            $rpv \leftarrow$ GetRearPageVer($pageBuf$);
        until $fpv == rpv$;
```

space and propagate them to the device promptly so that the offloaded computation can see the most up-to-date versions. In this paper, we propose a cooperative buffer manager (CBM) for the data freshness problem, in which the device maintains data consistency by cooperatively tracking the state changes in the host applications. We assume the host applications access on-disk data through a buffer manager, which is a common pattern found in many database systems [20], [21], [22]. In this section, we first introduce the basic operations provided by a buffer manager and its internal state transitions (Section III-A). Then, we discuss how our CBM tracks the state changes in the host buffer manager to provide data freshness guarantees to ISC operators (Section III-B and Section III-C).

A. Host Buffer Manager

Data processing systems such as DBMSs often forgo OS-provided caching services and re-implement buffer management for performance. The buffer manager acts as an intermediate layer between the foreground transaction processing and the backend storage, and handles data transfers between the storage device and host memory. It comprises a buffer pool, which contains a list of pages from the underlying data file. Each page is assigned a unique page identifier (PID), which also determines the physical location of a page on the storage device. To avoid double caching, the page file is opened with the `O_DIRECT` flag to disable caching at the file system level.

With the buffer manager, the foreground query processor requests data pages by providing the PIDs. When the buffer manager receives a request, it uses an index to translate the PID into the pointer to the desired data page. If the page has not been cached in the buffer pool (page miss), the buffer manager allocates a new page frame through the `mmap` system call and reads the data from storage into it. After acquiring a cached page from the buffer manager, the query processor may modify the page, and flag it as *dirty*. When the buffer pool is full and the requested page is not cached, the buffer manager evicts a *victim* page and replaces it with the requested page based on a page replacement policy. If the evicted page is dirty, it needs to be written back to the storage first.

One important aspect of buffer manager is synchronizing concurrent accesses to the same page. For example, the buffer manager may need to evict a page while it is being modified by the query processor. To prevent such data races, the query processor can *pin* a page so that the buffer manager will keep the page in memory until it is unpinned. After pinning the page, the query processor may also lock the page for exclusive or shared access.

Figure 2 shows the state transitions of data pages from the view of the host buffer manager. Initially, all data pages are in the `Evicted` state. When a page is accessed by the query processor, it is read into memory and pinned. When the query processor is done using a page, it is unpinned, and eventually evicted from the buffer manager. Note that to avoid data loss because of system failure, the buffer manager will append modifications to an on-disk write-ahead log (WAL) before data are flushed to the storage.

B. Device Mirror Cache

It can be seen from Figure 2 that whether the host buffer manager contains a more recent copy of a data page is closely related to its state. When a page is in the `Evicted` state, then the most recent version of the page must have been written back to the storage device. If a page is in the `Pinned` or `Unpinned` state, then the host caches a copy of the page, and we can use the dirty flag to determine whether this copy is more recent than that on the device. If we can replicate the *page state* from the host buffer manager to the device, then the device can decide whether to read the most up-to-date version of data from the host cache memory or the persistent data.

To this end, we maintain a mirror cache in the device DRAM to cooperatively track the states of pages in the host buffer manager. As shown in Figure 3, the mirror cache is an array of page descriptors. Each slot corresponds to a page that is present in the host buffer manager (*i.e.*, not evicted). The maximum number of page descriptors matches the capacity (number of pages) of the host buffer manager. Each page descriptor stores the PID, the state and the address of the copy in the host buffer manager for a cached page. The page state can be `Clean` or `Dirty`, based on the dirty flag managed by the host buffer manager. We also build an index that translates a PID into its slot in the page descriptor array for fast accesses. Algorithm 1 shows the overall process. For an ISC operator to read a data page into device DRAM, we first use the index to find the page descriptor for the user-provided PID. If the page is not found in the cache (*i.e.*, evicted from the host) or its page state is `Clean`, the device can safely read the most up-to-date version from the flash memory through the flash translation layer (FTL). Otherwise, the device reads the cached copy from the host memory using the address in the page descriptor.

For pinned pages, an exclusive lock only blocks accesses from other CPU threads but not concurrent reads from the device. If the device reads a page that is being concurrently modified by the host, it may read partial updates, leading to data inconsistency. We use a version-based optimistic read protocol for the device to detect inconsistent data caused by concurrent modifications. A pair of version counters, *i.e.*, *Front Version* and *Rear Version*, are stored at the beginning and the end of each page. When modifying a data page, the host thread first increments the rear version counter and then updates the page data. Finally, it increments the front version counter. The version counters are incremented during normal page locking/unlocking operations to minimize code modifications. After reading a data page from the host memory using direct memory access (DMA), the device compares the two version counters in the page. Mismatched versions indicate concurrent modifications from the host and thus the read must be retried.

C. Page State Piggybacking

With the mirror cache, the device can find and access the most recent copy of a data page without host intervention. One important question remains as to keep the mirror cache in synchronization with the host buffer manager. After all, we cannot let the host to actively propagate the page states to the device because it would generate one I/O request for every state change, offsetting all the benefits that a buffer manager brings. If we can piggyback on the I/O commands arising from cache management operations, the page state changes can be propagated to the device without issuing extra requests.

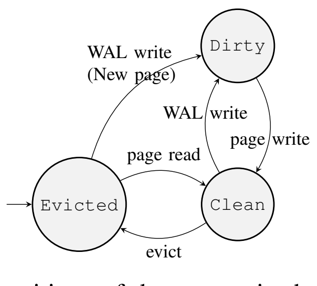


Fig. 4: State transitions of data pages in the mirror cache.

With a buffer manager, a host application mainly issues three kinds of I/O commands to the device: page reads, page writes and WAL writes. As shown in Figure 4, all pages initially start as `Evicted`. When the host buffer manager reads a page from the storage, it becomes `Clean` in the mirror cache as now the host memory contains an unmodified copy of the page. We can obtain the address to this cached copy in the host memory from the page read command because it also specifies the host address where the data should be written.

When the host application modifies a cached page, it needs to flush a log record to the device to protect the in-memory data. Because we assume a hybrid logging scheme, we can extract the affected PIDs from the log records and mark them as `Dirty` in the mirror cache. Furthermore, the host application may create a new page by extending the data file. In this case, we allocate a new page descriptor for the new page and mark it as `Dirty` directly. A page write command is only issued as the result of periodic flushing or page eviction; either way, we mark the page as `Clean` in the mirror cache as the most recent data have been written back to the device. When the host buffer manager replaces a page, it reuses the victim buffer to read the data of the new page. Thus, we can use the host buffer address provided in the page read commands to check if any entry in the mirror cache has been evicted from the host buffer manager. Specifically, we build an index in the device controller memory to map host buffer addresses to page descriptors. Upon read commands, we check the mirror cache to see if any page descriptor has the same host buffer address as the one provided in the command. If so, the corresponding host page has been evicted and reused and the page descriptor should be removed from the mirror cache.

IV. Evaluation

A. Experimental Setup

1) Hardware platform: We implement the proposed cooperative buffer manager based on the SoftSSD development board [23]. The development board is connected to the host PC via PCIe and used as a standard NVM Express (NVMe) [24] block device. The device hook functions are called when the FTL receives NVMe read/write commands from the host to update the mirror cache state. Overall, the mirror cache and the hook functions for the device FTL can be implemented in ~ 250 lines of code. Table II summarizes the configurations of the hardware evaluation platform.

TABLE II: Hardware evaluation platform configurations.

Host	CPU	Intel® Core™ i7-8700 @3.6GHz (6C12T)
	Memory	16GB
	Kernel	Linux 6.3.1
Device	SoC	Xilinx® Zynq™ Ultrascale+™ ZU7EG
	CPU	4-core Arm® Cortex®-A53 @1.2GHz 2-core Cortex-R5 @533MHz
	Memory	8GB DDR4
	Storage	4× 128GB MLC NAND
	Host interface	NVMe over PCIe Gen3 x8

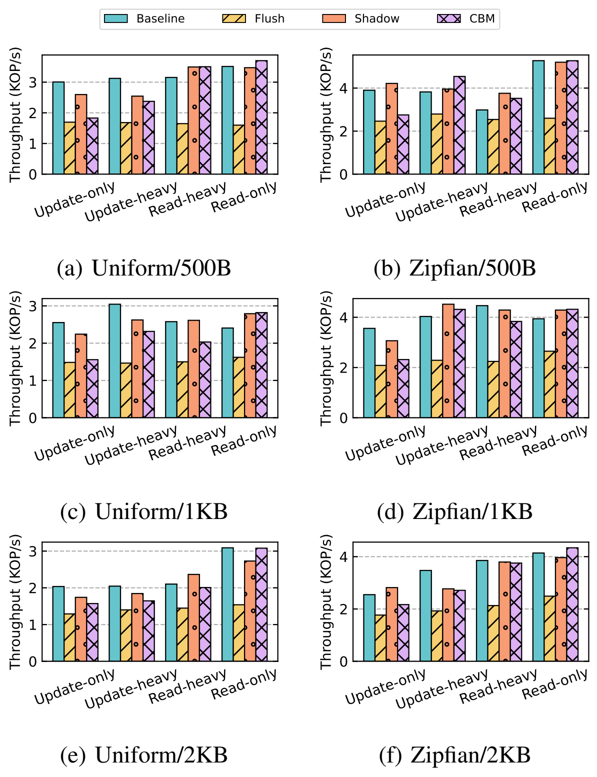


Fig. 5: Host throughput (higher is better) of YCSB workloads by key distribution/update value size.

2) Software environment: On the host side, we implement a userspace NVMe driver to communicate with the device and a page cache based on it. The page cache manages data as 16KB pages and directly maps the page IDs to device logical block addresses. To differentiate between data page and log writes, we reserve a special logical block address range as the WAL region. With the page cache, we implement a key-value (KV) store based on B+-tree [25]. The size of B+-tree nodes matches the page size (16KB) and we use the page IDs as the node IDs. Each node contains a sorted list of pivot keys (internal nodes) or KV records (leaf nodes). The KV store supports common operations such as `get`, `put` and `update` with variable-length string keys and values, where the point lookup operation is also implemented as an ISC operator and integrated with the device FTL.

B. Evaluation on YCSB Workloads

We use the YCSB benchmark [26] to evaluate the performance of the proposed CBM. With YCSB's data model, each key has multiple fields that are concatenated to form the string values. We use 24B keys and small (500B), medium (1KB), and large (2KB) values. The core YCSB workload consists of a load phase and a run phase. In the load phase, we populate an empty B+-tree database with 2M KV pairs with 1KB values ($\sim$ 3.7GB). The run phase simulates different workloads with an OLTP thread and an OLAP thread running in parallel. The OLTP thread executes update and read operations with a predefined read-write ratio on cached data pages. We use four workloads with different read-write ratios: 1) **Read-only**, 2) **Read-heavy** (70%R/30%W), 3) **Update-heavy** (30%R/70%W), and 4) **Update-only**. Meanwhile, the OLAP thread continuously sends ISC B+-tree point lookup requests to the device. The keys used by the operations during run phase are generated with a specific key distribution. The *Uniform* distribution selects each key in the database with equal probability. The *Zipfian* distribution simulates a skewed data access pattern where a portion of the keys is accessed more frequently than the others.

We compare our proposed CBM with three methods: `Baseline`, `Flush`, and `Shadow`. The `Baseline` method does not perform any synchronization for data freshness and the host-side and device-side ISC operations run independently from each other. The capacity of the host buffer manager is set to be large enough so that no dirty pages will be written back to the device. The `Flush` method is based on λ-IO [5], in which the OLAP thread identifies and flushes all dirty pages in the host buffer manager that contain committed updates before offloading a read operation. The `Shadow` method is based on update-aware NDP [19], which accumulates all updated KV pairs in a shadow skiplist-based table in the host memory similar to an LSM tree [27], [28], [29]. Before offloading a read operation to the device, the OLAP thread serializes the shadow memory table into a Sorted String Table (SST), which comprises data blocks with the actual KV pairs sorted by keys, and a block-level index. The SST is transferred to the device memory, which will be utilized by the device upon ISC B+-tree lookup requests to check if a more recent value for the searched key exists. In this experiment, we set the capacity of the shadow memory table to 128KB. For all methods, the run phase stops after 200K read requests have been offloaded to the device by the OLAP thread and completed.

1) Host-side performance: We first present the throughput and average operation latency of the host-side OLTP thread with different update value sizes during the run phase in Figure 5 and Figure 6, respectively. As shown in the figures, the `Flush` method has negative impacts on the host-side throughput and latency because the OLAP thread needs to lock the host buffer manager to find dirty pages that need to be flushed, which in turn blocks the operations from the OLTP thread. Meanwhile, `Shadow` and `CBM` achieve similar throughput and latency to `Baseline`. For updates, CBM utilizes the existing log write commands to propagate cache state changes to the device without generating extra I/O commands compared to `Baseline`. For the Zipfian workload, the host buffer manager can better exploit the data locality to reduce the number of pages read into host memory so the update throughput is better than that for the Uniform workload.

2) Device-side ISC point lookup latency: Figure 7 shows the cumulative distribution function (CDF) of the end-to-end latency of KV point lookup requests offloaded to the device, with concurrent OLTP workloads. As shown in Figure 7, all methods except `Flush` achieve similar lookup latency for the Read-only workload because no synchronization is needed. For `Flush`, the OLAP thread still needs to check if there are dirty pages that need to be flushed, which leads to a higher

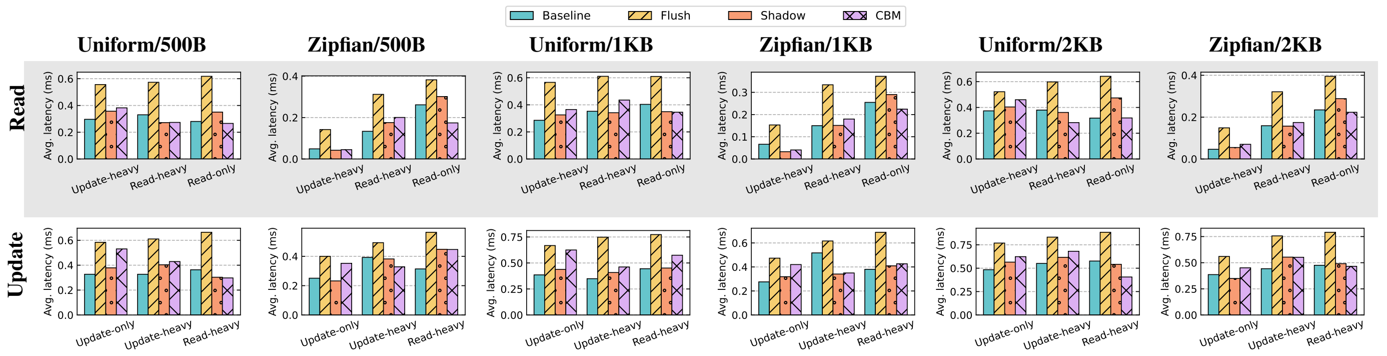


Fig. 6: Average host read/update operation latency (lower is better) of YCSB workloads by key distribution/update value size.

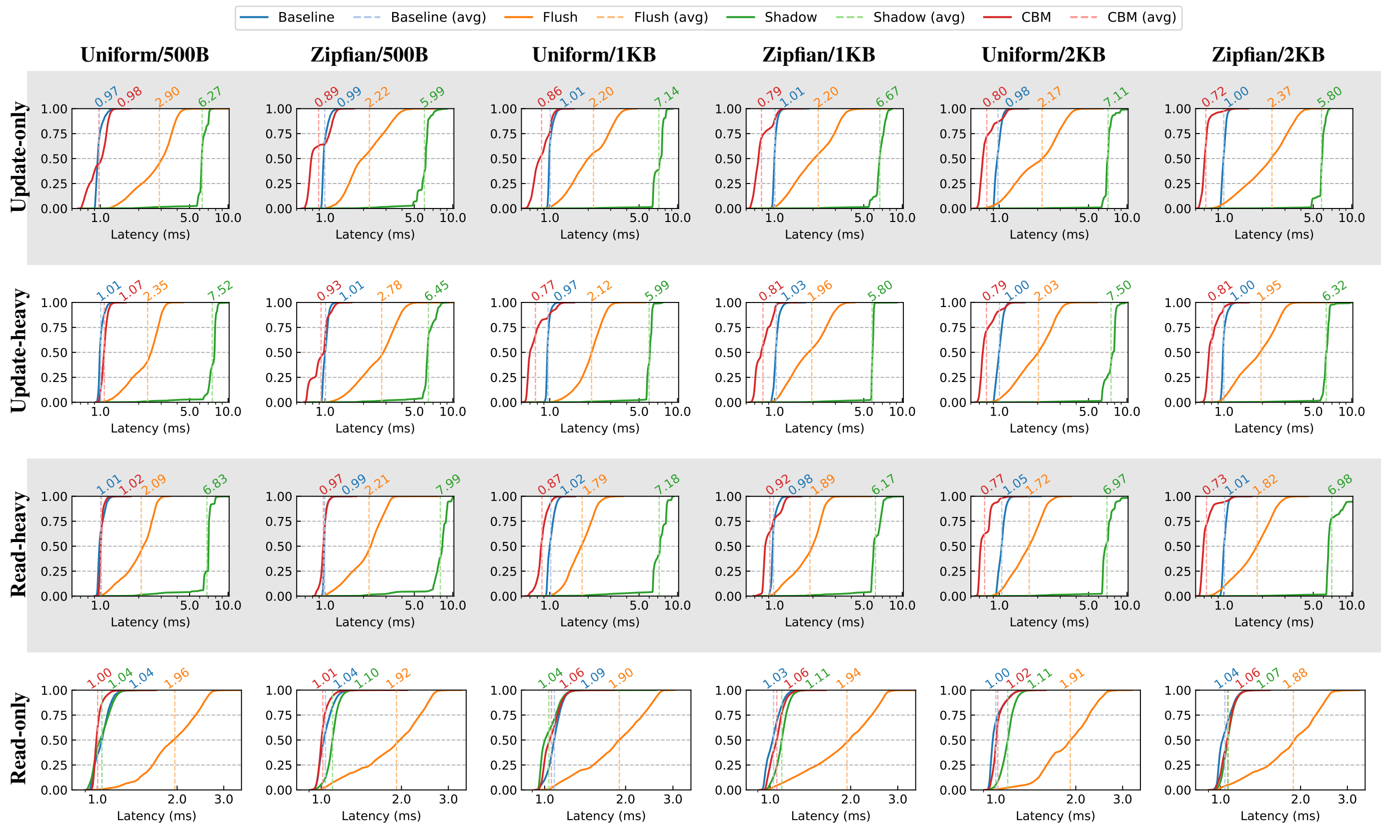


Fig. 7: Cumulative distribution function (CDF) of end-to-end in-storage KV point lookup operation latency under different YCSB workloads by key distribution/host update value size. The Y-axis shows the cumulative probability.

device-side lookup latency even for the Read-only workload. When there are concurrent updates from the host OLTP thread, `Shadow` needs to serialize the shadow memory table and transfer it to the device. Meanwhile, the ISC operator needs to search the key in the shadow data structure and subsequently the persistent B+-tree if an updated value is not found. These extra steps lead to the higher device-side lookup latency of `Shadow` under all workloads other than Read-only. CBM, instead, allows the host buffer manager to keep dirty pages when issuing ISC requests and lets the device decide where to retrieve the data. Thus, it can achieve shorter device-side latency. Furthermore, `CBM` improves the point lookup latency compared to `Baseline` under Update-heavy and Update-only workloads because it can alternatively read dirty pages from the host memory via PCIe, which reduces the flash channel contention in the SSD.

V. Conclusion

In this paper, we propose a cooperative buffer manager for in-storage computing to transparently provide data freshness guarantee to host applications. The CBM allows the device to monitor the state of the host buffer manager and retrieve the updates from the host memory on demand. We implement our method based on a real hardware platform and perform evaluation with a B+-tree-based key-value store. Experiments show that our method can guarantee data freshness and reduce the end-to-end latency for offloading ISC tasks.

References

[1] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. Near-Data Processing: Insights from a MICRO-46 Workshop. *IEEE Micro*, 34(4):36–42, July 2014.

[2] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart SSDs: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, June 2013.

[3] Zhenyuan Ruan, Tong He, and Jason Cong. INSIDER: Designing In-Storage Computing System for Emerging High-PerformanceDrive. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 379–394, 2019.

[4] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *ACM SIGOPS Operating Systems Review*, 32(5):81–91, October 1998.

[5] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. λ-IO: A Unified IO Stack for Computational Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, 2023.

[6] Yifan Qiao, Xubin Chen, Ning Zheng, Jiangpeng Li, Yang Liu, and Tong Zhang. Closing the B+-tree vs. LSM-tree Write Amplification Gap on Modern Storage Hardware with Built-in Transparent Compression. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 69–82, 2022.

[7] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 591–605. Association for Computing Machinery, New York, NY, USA, March 2020.

[8] Devesh Tiwari, Simona Boboila, Sudharshan Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 119–132, 2013.

[9] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A Framework for Near-Data Processing of Big Data Workloads. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 153–165, June 2016.

[10] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, August 2017.

[11] Simona Boboila, Youngjae Kim, Sudharshan S. Vazhkudai, Peter Desnoyers, and Galen M. Shipman. Active Flash: Out-of-core data analytics on flash storage. In *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[12] Tobias Vinçon, Arthur Bernhardt, Ilia Petrov, Lukas Weber, and Andreas Koch. nKV: Near-data processing with KV-stores on native computational storage. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, pages 1–11, New York, NY, USA, June 2020.

[13] Shuyi Pei, Jing Yang, and Qing Yang. REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage. *ACM Transactions on Storage*, 15(1):7:1–7:24, March 2019.

[14] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced SQL offloading. *Proceedings of the VLDB Endowment*, 7(11):963–974, July 2014.

[15] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. YourSQL: A high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, August 2016.

[16] Andreas Becher, Stefan Wildermann, and Jürgen Teich. Optimistic regular expression matching on FPGAs for near-data processing. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, pages 1–3, New York, NY, USA, June 2018.

[17] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. POLARDB Meets Computational Storage: Efficiently Support Analytical Workloads in Cloud-Native Relational Database. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 29–41, 2020.

[18] Antonio Barbalace, Martin Decky, Javier Picorel, and Pramod Bhatotia. blockNDP: Block-storage Near Data Processing. In *Proceedings of the 21st International Middleware Conference Industrial Track*, Middleware '20, pages 8–15, New York, NY, USA, December 2020.

[19] Tobias Vinçon, Christian Knödler, Leonardo Solis-Vasquez, Arthur Bernhardt, Sajjad Tamimi, Lukas Weber, Florian Stock, Andreas Koch, and Ilia Petrov. Near-data processing in database systems on native computational storage under HTAP workloads. *Proceedings of the VLDB Endowment*, 15(10):1991–2004, June 2022.

[20] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-Memory Assisted Buffer Management. *Proceedings of the ACM on Management of Data*, 1(1):7:1–7:25, May 2023.

[21] Ling Feng, Hongjun Lu, and Allan Wong. A study of database buffer management approaches: Towards the development of a data mining based strategy. In *SMC'98 Conference Proceedings. 1998 IEEE International Conference on Systems, Man, and Cybernetics (Cat. No.98CH36218)*, volume 3, pages 2715–2719 vol.3, October 1998.

[22] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 2195–2207, New York, NY, USA, June 2021.

[23] Jin Xue, Renhai Chen, and Zili Shao. SoftSSD: Software-defined SSD Development Platform for Rapid Flash Firmware Prototyping. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 602–609, October 2022.

[24] NVM Express Base Specification 2.0. page 452.

[25] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, Indianapolis, Indiana, USA, June 2010.

[27] Yoshinori Matsunobu, Siying Dong, and Herman Lee. MyRocks: LSM-tree database storage engine serving Facebook's social graph. *Proceedings of the VLDB Endowment*, 13(12):3217–3230, August 2020.

[28] Haoyu Huang and Shahram Ghandeharizadeh. Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, pages 749–763, New York, NY, USA, June 2021.

[29] Chen Luo and Michael J. Carey. LSM-based storage techniques: A survey. *The VLDB Journal*, 29(1):393–418, January 2020.