UHS: An Ultra-fast Hybrid Storage Consolidating NVM and SSD in Parallel

Qingsong Zhu*, Qiang Cao*✉, Jie Yao†
*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology,
†School of Computer Science and Technology, Huazhong University of Science and Technology,
✉Corresponding Author: caoqiang@hust.edu.cn

***Abstract*—Non-Volatile Memory (NVM) with persistency and near-DRAM performance has been commonly used as first-level fast storage atop Solid-State Drives (SSDs) and Hard Disk Drives (HDDs), constituting classic hierarchy architecture to achieve high cost-performance. However, such NVM/SSD tiered storage overuses primary NVM with limited actual performance and under-utilizes secondary SSD with increasing bandwidth. Besides, NVM and SSD exhibit distinguished I/O characteristics, but are complementary for different I/O patterns. This motivates us to design a superior hybrid storage to fully exploit NVM and SSD simultaneously.**

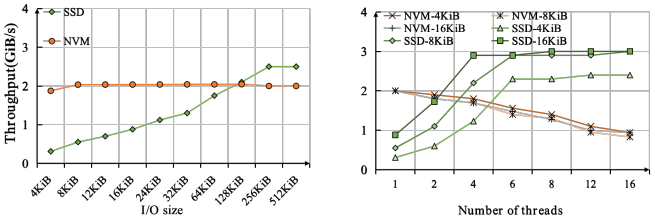
In this paper, we propose UHS, an Ultra-fast Hybrid Storage consolidating NVM and SSD to reap their own merits with key enabled techniques. First, UHS builds a uniform yet heterogenous block-level storage view for the upper applications, e.g., file systems or key-value stores. UHS provides static address-mapping to explicitly partition the global block-space into coarse-grain NVM-zones and SSD-zones, which mainly serve the metadata and file data respectively. Second, UHS presents a fine-grain request-level NVM buffer to dynamically absorb small file-writes in runtime and then migrates them to the SSDs in the background. Third, UHS designs I/O-affinity write allocation and hash-based buffer indexing to trade off write gain and read cost of the NVM-buffer. Finally, UHS designs a multi-thread I/O model to take full advantage of parallelism in both NVM and SSD. We implement UHS and evaluate it under a variety of workloads. The experiments show that UHS outperforms SSD, NVM, Bcache-writeback (representative hierarchy storage), and Device-Mapper (state-of-the-art hybrid storage) up to 8X, 1.5X, 3.5X, and 6X respectively.

***Index Terms*—Hybrid Storage, Non-Volatile Memory, I/O Parallelism**

I. Introduction

Non-Volatile Memory (NVM) [1] is considered as persistent memory blurring the conventional boundary between memory and storage, attracting extensive attention from both academy and industry. Traditional storage stack originally designed for Hard Disk Drives (HDDs) and Solid-State Drives (SSDs) cannot sufficiently exploit byte-addressability and fast persistency of NVM. Therefore, recent many works redesign novel data structures [2], NVM-aware file systems [3] [4], and NVM-aware Key-Value (KV) stores [5], to make full use of NVM using byte-accessed interface (e.g., load/store) and dedicated I/O stack (e.g., PMDK). Mature file systems as EXT4 [6] and XFS [7] employ direct access (*DAX*) [8] and memory-map mechanism.

Meanwhile, SSDs have been evolving to increase both performance and capacity at a decreasing cost. An intuitive idea is to combine NVM and SSD to achieve a higher cost-performance than either of them. Using a classic storage hierarchy architecture, fast yet small-capacity NVM absorbs all writes and most reads as the first-level cache/buffer, while slow but large-capacity SSDs serve the missed reads and writes evicted from the NVM in the background. Following this architecture, recently, tiered file systems, such as Strata [9] and Ziggurat [10], are designed to buffer hot or recent files in the NVM.



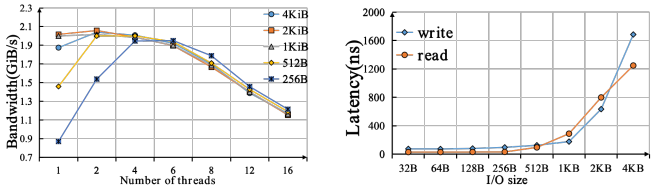
(a) Throughput with single thread (b) Throughput with different threads and I/O Size

Fig. 1: **a)**Write Throughput vs I/O size ; and **b)**Write Throughput vs thread count on Optane Persistent Memory and SSD

Recent researches and our experiments observe that commodity NVM, i.e., Intel Optane DC Persistent Memory Model [11], exhibits a significant performance advantage only for small-sized reads/writes compared to NVMe-based SSDs. However, for large-size writes (more than 128KiB), the fast SSDs with high internal-parallelism exhibit an almost similar even higher I/O bandwidth than NVM, as shown in Figure 1(a). Besides, Figure 1(b) shows that NVM has bounded IO-thread scalability with a peak write throughput at 4 threads. SSDs have higher thread-level parallelism and write throughput. Therefore, NVM does not exhibit a significant advantage in the write performance over fast SSDs. With the narrowing performance gap between NVM and SSD, such NVM-SSD hierarchy storage inevitably overuses NVM but under-utilizes fast SSDs.

To unleash their own potentials simultaneously, this paper proposes UHS, an Ultra-fast Hybrid block-level Storage consolidating NVM and SSD to serve the upper applications in parallel. The key idea of UHS is strategically allocate favorite I/Os to NVM or SSD. To this end, first, UHS offers a global and uniform block-level storage space for upper storage software (e.g., file system) upon NVM-zone and SSD-zone while statically mapping NVM-affinity data (e.g., metadata) to the underlying NVM-zone. Second, UHS provides an NVM buffer to dynamically absorb small SSD-targeted writes, and then migrates them to the SSD. However, this out-of-place write mode also introduces checking buffer before an SSD-access and extra migration cost. To address the challenge, UHS designs

978-3-9819263-7-8/DATE23/© 2023 EDAA



(a) Throughput of NVM (b) Latency of NVM

Fig. 2: **a)** Write Throughput vs thread count; b) Latency vs I/O size on Optane NVM using FIO [12] with libpmem engine.

dynamic NVM-buffer accessing, hash-based buffer indexing, fast migration, and crash recovery to fully utilize the NVM-buffer. Finally, UHS provides a parallel I/O-thread scheduling mechanism to harness the limited parallelism of NVM.

UHS offers a hybrid block-level storage view to hide different I/O accessing characteristics and I/O stacks of NVM and SSD. This means mature storage software, such as file systems and KV stores, can be portable to UHS effectively, thus taking full advantage of both NVM and SSD without major modification. To the best of our knowledge, UHS is the first ultra-fast hybrid block-level storage built upon NVM and SSD in parallel way.

The major contributions of this work are:

- We present a static address-mapping with a unified data layout but explicitly exposes its NVM block-space, as well as a dynamic address-mapping to absorb small writes using the NVM-buffer in runtime;
- We design a parallel I/O scheduling and migration strategy for full I/O exploitation of NVM and SSD simultaneously;
- We implement UHS and evaluate it under a variety of workloads. The experiments show that UHS outperforms SSD, NVM, Bcache-writeback (representative hierarchy storage), and Device-Mapper (state-of-the-art hybrid storage) up to 8X, 1.5X, 3.5X, and 6X respectively.

II. Background and Related Work

A. NVM and SSD

Non-Volatile Memory (NVM) [1] [**?**] promises near-DRAM performance and behavior (byte-addressability) but persistency, will become an integral part of future superior storage.

However, the experiment on Intel Optane DC Persistent Memory [11] shows that, its write and read bandwidth in App-direct mode is about 2.3GB/s and 6.6GB/s respectively, which is much lower(10x-20x) than the write and read bandwidth of DRAM. The write bandwidth is even lower than Solid-State Drive (SSDs) (about 3.5GB/s) when the write size exceeds 128KiB. Besides, NVM has limited I/O parallelism as shown in Figure 2(a). NVM can reach the peak throughput with about four threads. When the number of threads increases, the aggregate throughput even decreases significantly. Besides, NVM exhibits a remarkable read-write asymmetry where reads outperform writes by 2∼3X in the I/O delay and throughput.

On the other hand, NAND-based SSDs have been increasing their capacity and performance over last decade, gradually replacing traditional Hard-Disk Drives (HDDs). Recently, NVMe-based SSDs exhibit dozens of μ I/O latency and GB/s-level bandwidth. Certainly, SSDs have inherent drawbacks such as limited Program/Erase (P/E) cycles, out-of-place updates, heavy garbage collection (GC), and severe read/write asymmetry.

In summary, NVM still exhibits superior byte-level accessibility, compared to SSDs and HDDs, as shown in Figure 2(b). However, for large reads/writes (more than 128KiB), fast SSDs have a closed even higher throughput. Therefore, NVM and fast SSD are very complementary in different sized I/Os and internal parallelism.

TABLE I: Characteristics of Representative NVM Storage.

	Ext4-DAX	NOVA	Strata	Ziggurat	FR	UHS
NVM-SSD Hybrid	No	No	Yes	Yes	Yes	Yes
Abstract-Level	File System	File System	File System	File System	Block	Block
Buffer Acceleration	No	No	No	Yes	No	Yes

B. Related Work

The emergence of NVM reshapes traditional storage I/O stack. We summarize the characteristic of representative NVM storage systems in Table I in three dimensions as NVM-SSD hybrid, abstract-level, and buffering-acceleration. A body of research to exploit the characteristics of NVM is revising or redesigning NVM-aware file system, for example, Ext4-DAX [8] and NOVA [13].

Considering NVM is fast, small and expensive while SSD is relatively slow, large and cheap, prior works also design file systems and cache/buffer using NVM-SSD tiered storage. Strata [9] writes all data directly to the upper-layer NVM, and then migrates cold data to the large-capacity SSD. Ziggurat [10] further selectively sends asynchronous or large write operations to the secondary storage device according to dedicated predictors. Besides, the First Responder(FR) [14] adds the NVM as the cache of existing file system to absorb requests at the topmost layer of the I/O stack.

Bcache [15] is a Linux block-cache and supports fast storage as cache of multiple slow storages, which is a typical hierarchical storage architecture. Bcache does not support DAX-mode and can be only accessed via traditional I/O stacks. Device Mapper [16] is an address-mapping from logical storage to physical storage but does not employ NVM directly.

Existing works largely neglect the fact that SSD outperforms the NVM in large I/O performance and thread-level scalability. More importantly, the classic storage hierarchy makes NVM overloaded while the I/O capacity of SSD is largely wasted.

C. Motivation

Note that NVM and SSD have their own favorite I/O patterns. For example, file-system metadata updates with small size (e.g., smaller than 4KiB) are notoriously inefficient for HDDs and SSDs, but are friendly to NVM. In contrast, large file data stored upon the SSDs requires cost-performance. However, existing both NVM-alone and NVM-SSD storage cannot reap their own merits of NVM and SSD simultaneously without major modification for existing storage applications (e.g., mature file system) .

Nowadays, both NVM and SSD are off-the-shelf and easy to be installed within a machine. Full exploitation for NVM-SSD hybrid storage is valuable but still challenging.

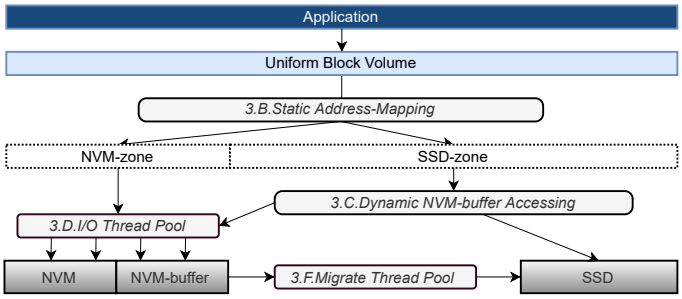


Fig. 3: Architecture of UHS

III. Design

A. Overview

To fully exploit NVM and fast SSD, we propose UHS, a heterogeneous yet uniform block-level storage layer for existing storage applications (e.g., EXT4 and key-value stores). The architecture of UHS is shown in Figure 3. First, UHS provides Static Address-Mapping for NVM-affinity data of upper applications. Second, UHS offers an NVM-buffer to conditionally absorb small SSD-writes by Dynamic NVM-buffer Accessing mechanism. Third, UHS proposes an I/O-thread scheduling to decouple user and I/O threads to exploit the internal parallelism of NVM and SSD in parallel. Last, UHS strategically migrates all buffered data to the SSDs. Next, we will elaborate on these techniques.

B. Static Address-Mapping

UHS provides a uniform block space but can statically map a given block-range to an NVM-zone using predefined configuration like DM-table, for instance, the NVM-zone from block 0 to block N in the global space. Existing file systems, such as Ext4 [6] and F2FS [17], have their specific data-layout defining the areas of all metadata and file data. For example, when an Ext4 file system is formatted, its metadata as Superblock, Block Bitmap, Inode Bitmap, and Inode Table are deterministic. UHS can simply map its upper file system's metadata-areas into the NVM-zone and the other data into the SSD-zone. As a result, UHS can send a write to NVM when its address belongs to the predefined NVM-address-range. Otherwise, the write is delivered to the dynamic NVM-buffer accessing module.

The metadata generally are frequently accessed in small-size, such as 4B for bitmap and 128B for Inode, well benefiting from NVM property. With legacy Linux I/O stack, a block-layer as UHS only receives at least 4KiB-sized bio requests when accessing metadata, which is sub-optimal I/O granularity for NVM but is still beneficial. Emerging NVM-based file system with byte-addressability can take more advantage of UHS. The build-in static mapping avoids costly behavior-prediction and inaccuracy in runtime such as Ziggurat [10] and implicit metadata caching.w

C. Dynamic NVM-buffer Accessing

According to Static Address-Mapping, all non-NVM requests should be sent to the SSDs. However, small SSD reads/writes generally cause high delay, I/O amplification, and wear-out, but can well be handled by NVM. Therefore, UHS also offers a dedicate NVM-buffer to serve small SSD-targeted requests in running time.

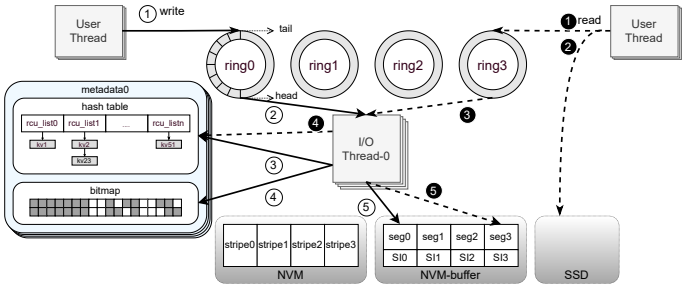


Fig. 4: UHS Write/Read with NVM-buffer.

This small-write-buffering idea is relatively simple and friendly to write, but also causes extra access-ahead buffer-checking due to out-of-place writes in the NVM-buffer. For a read and write, it is necessary to first determine where its new value is, i.e., NVM or SSD, which is in the critical path. To mitigate the extra overhead, UHS introduces a hash-based indexing mechanism to fast determine whether the new value of requested block is in the NVM-buffer.

The architecture of NVM-buffer is shown in Figure 4. The NVM-buffer contains a set of segments (e.g., 4 MiB each segment) and segment information, called **SI**. An SI records the segment state (e.g., clean or used), valid bits, hash-tables, bitmap, flag and address-range of SSD. The hash-table accelerates address-retrieve; the bitmap records whether the block is used, and the flag indicates whether current segment is in migrating. An NVM-segment buffers the SSD blocks of a predefined range of the SSD-zone. UHS records the address of an SSD-block as key and its buffered address in the NVM-buffer segment as value. UHS further adopts a chain method to resolve the hash-conflict when frequently inserting and deleting. UHS organizes all key-value pairs with **rcu_list**, which uses Read-Copy-Update to avoid multi-thread contention.

To speed up, UHS buffers all SIs and synchronizes them between NVM and DRAM. UHS also restricts the NVM-buffer size (e.g.,16 MiB by default) to reduce the range and delay of looking-up. Besides, UHS actively migrates buffered data to the SSD under light load, or reads/writes SSD directly under heavy small-IO load, avoiding frequent migration.

Static NVM-zone and Dynamic NVM-buffer serve NVM-affinity data of deterministic application-aware and indeterministic runtime-I/Os respectively, which are complementary to fully leverage NVM. UHS can adjust the size of NVM-buffer and the threshold of buffered-writes to effectively control load intensity upon NVM and SSD at an optimal balancing point.

D. Parallel I/O-thread scheduling

NVM has a limitation of parallelism, especially for write. Figure 2(a) shows that the NVM write performance peaks at about 4 threads with poor scalability. However, the number of user-threads is uncontrolled and varied. Therefore, UHS uses a parallel I/O-thread scheduling model, which **1)** decouples I/O-thread from user-thread and uses an **I/O Thread-Pool** with fixed 4 I/O-threads to read and write, which fits the limited scalability of NVM, **2)** divides NVM and NVM-buffer into a

Algorithm 1 Dynamic NVM-buffer Write

Small Write

```
1: idx ← hash(offset)
2: send request to I/O-thread[idx]
3: wait(request completes) /*notified by line-15*/
   In I/O-thread[idx]
4: if block in segments[idx] then
5:     write request
6:     update metadata
7: else
8:     blk ← a valid block in segments[idx]
9:     if blk == −1 then
10:        migrate segments[idx]
11:    end if
12:    write request
13:    record blk in metadata
14: end if
15: notify request completes
```

Large Write

```
1: if NVM-buffer is not clean then
2:     send sub_request to all I/O-threads
3: end if
4: write data to SSD-zone
   In I/O-thread
5: for block in request do
6:     if block in current segment then
7:         clear block
8:     end if
9: end for
```

set of chunks, **3)** binds an I/O thread to a specific chunk to avoid concurrent access and out-of-order of *associated requests*.

The *associated requests* that are generated by the same user-thread and access the same address in three orders as read after write (RAW), write after write (WAW), and write after read (WAR). For example, user-thread reads block-n first and then write block-n, which are WAR *associated requests*. If they are sent to different I/O-threads and the write incurs before its associated read finally, the read value could be old. Therefore, before sending requests to I/O-thread, UHS hashes their offsets. Afterwards, the user-thread sends them to the target ring buffer and the I/O-thread will poll to retrieve them in the original order. For SSD-writes, the user-thread can perform their own SSD-I/Os in parallel to take full advantage of the capabilities of both devices.

E. Write and Read

Next, we describe the entire process of small SSD-writes, large SSD-writes, and SSD-reads, respectively.

Small SSD-Write. Algorithm 1 describes the small SSD-write process as shown in Figure 4: ① User-thread hashes offset of an incoming write to get the $index$, sends the write as a request to ring buffer[$index$] and waits for complete. ②I/O-thread[$index$] polls the bound ring buffer[$index$] to retrieve the request, and ③accesses metadata[$index$] to search the hash-table to determine whether the block exists in segment[$index$]. ④ Otherwise, try to get a free block by searching bitmap, when there is no free block, UHS waits for the segment[$index$] to enforce migrating. ⑤ I/O-thread[$index$] writes data to the

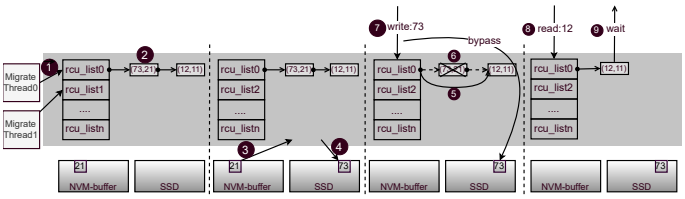


Fig. 5: Migration from NVM-buffer to SSD.

segment[$index$], updates the metadata[$index$], adds the target and buffered address to hash-table and notifies user-thread that the write has completed.

Large SSD-Write. For large SSD-write, UHS processes it as shown in Algorithm 1: **1)** User thread determines whether the NVM-buffer's state is clean. **2)** Otherwise, user thread sends $sub\_request$ with CLEAR flag to each I/O-thread. **3)** User thread writes data to the SSD-zone. **4)** Each I/O thread traverses $sub\_request$ to invalidate the block in current segment and clears the metadata.

SSD-Read. The strategy used by UHS to handle the SSD-read includes the following steps: ❶ User-thread splits SSD-Read to $sub\_request$ with 4KiB block size, ❷ sends $sub\_request$ to I/O-threads and reads from SSD-Zone to ssd_{buf} in parallel. ❸ I/O-thread[$index$] looks up metadata[$index$] to determine whether the target block is buffered in segment[$indx$]. ❹If so, I/O-thread[$index$] reads the block from segment[$index$] to nvm_{buf}. When all $sub\_requests$ have done, the I/O thread which completes the last $sub\_request$ notifies the user-thread to combine the ssd_{buf} and nvm_{buf}, and then return.

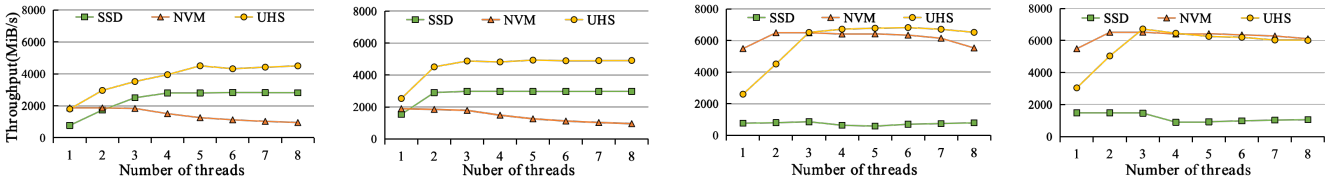
F. Migration

UHS absorbs small SSD-writes in the NVM-buffer, and then migrates them to SSD at light load, or out-of-space in a segment. UHS also creates a **Migrate Thread-Pool**. When migrating, as shown in Figure 5, each migrate-thread in thread-pool ❶ selects a hash-table entry, which contains a rcu_list, to get the key-value, ❷ traverses the rcu_list and retrieves the key-value, whose key is the target address in SSD, and the value is the buffered address in NVM-buffer, ❸ reads block from segment in NVM-buffer and ❹ writes it to the SSD, ❺ removes the key-value from hash-table and ❻ frees it atomically. When the migration ends, UHS will clear the bitmap and valid bits. Additionally, UHS sets the state of segment as 'clean', thus avoiding extra indexing for reads and large writes.

In order to relieve the impact on user read and write, we use rcu_list to organize the key-value, which ensures the correctness of concurrent read and write on migrating. When a request comes, ❼ if the target addresses are not in the hash-table, the accessed blocks are independent of the migrated data and can be bypassed to the SSD directly. ❽ Otherwise, the request is blocked ❾ until all accessed blocks of the request have been migrated.

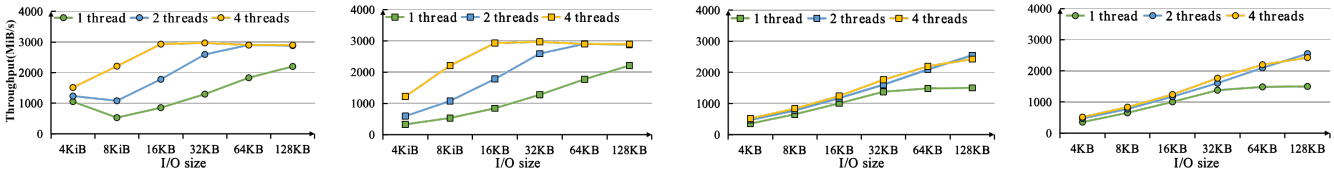
G. Crash Consistency and Recovery

UHS ensures the consistency when the system crashes or powers off. UHS synchronously updates SIs of DRAM and NVM-buffer, which hardly degrades the performance due to



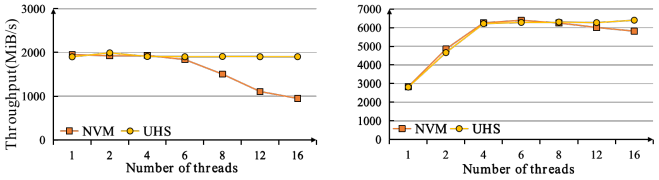
(a) 4KiB + 8KiB Write (b) 4KiB + 32KiB Write (c) 4KiB + 8KiB Read (d) 4KiB + 32KiB Read

Fig. 6: The write and read throughput as thread count increases with 1:1 Mixed I/O sizes.



(a) UHS Write (b) SSD Write (c) UHS Read (d) SSD Read

Fig. 7: The write and read throughput on the SSD-zone of UHS and SSD under with different thread-count and request-sizes.



(a) UHS and NVM Write (b) UHS and NVM Read

Fig. 8: The average write and read throughput on the NVM-zone of UHS and NVM with threads increasing.

fast-persistence of NVM. Besides, UHS invalidates the SI after the migration is completed. In case of crashing, UHS can recovery the NVM-buffer by scanning all SIs on the NVM. Therefore, UHS has a same crash-consistency level as traditional block-volumes provided.

IV. Evaluation

All experiments run on a dual-socket Intel Xeon E5 server with 16 physical cores at 2.30GHz, 256 GB DRAM, and a 128GB Intel Optane DC PMM. The operating system is Ubuntu 20.04 with Linux kernel 5.4.0-107. We compare UHS with NVM and SSD respectively. We also test two typical hybrid storage architectures, Bcache [15] and Device-Mapper [16]. In addition, we implement UHS without the NVM-buffer, called UHS-NB, to compare with UHS to verify the effectiveness for small write. Note that current UHS is implemented as a block-level storage in user-space and cannot be directly used by existing kernel-based file system and file-system-based applications. We use blktrace [18] to capture the block I/O events from Ext4, and then replay the traces as the workloads.

A. Microbenchmarks

We measure the read/write throughput of UHS, NVM, and SSD under FIO. For NVM, we use libpmem [19] of PMDK to directly access NVM through mmap. For SSD, we access the device file with the DIRECT_IO flag, like /dev/nvmen1, to avoid the impact of the upper cache and file system.

Throughput Figure 6 shows the average write and read throughput as thread count increasing with 1:1 Mixed I/O size, respectively on the NVM, SSD and UHS. With UHS, the 4KiB writes are stored in the NVM-zone or the NVM-buffer and the writes larger than 4KiB are directly written into SSDs. The I/O pattern with 4KiB+xKiB exploits the overall effect of NVM and SSD in UHS. For write, the throughput of UHS is always higher than NVM and SSD, and is approximately equal to their sum about 4.9GiB/s, which is **1.4x** and **1x** higher than NVM and SSD, respectively. For read, when the number of threads is less than 3, the throughput of UHS is lower than NVM because the read throughput of NVM is not saturated and SSD is much slower than NVM. However, when the number of threads exceeds 4, the throughput of UHS almost saturates the sum of the bandwidth of NVM and SSD.

Figure 7 shows the average write and read throughput of SSD-zone and SSD with different threads under varying I/O sizes. Specially, when the I/O size is 4KiB, the write throughput of SSD-zone is about **1.5x**∼**3x** higher than SSD. Because UHS uses the NVM as the NVM-buffer to absorb the small writes to SSD-zone. This manifests the advantage of the NVM-buffer.

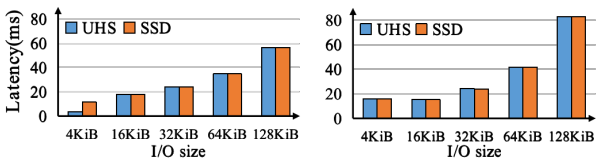
Figure 8 shows the average write and read throughput of NVM-zone. We test NVM-zone with 4KiB I/O size under different numbers of threads. Compared with the NVM, the NVM-zone has higher parallelism on write. When the number of threads is less than 4, the throughput of NVM-zone is equal to NVM. However, with the number of threads increasing, the write throughput of UHS does not drop significantly like NVM due to parallel I/O-thread scheduling, and is up to **2x** higher than NVM with 16 threads.

Latency Table II shows the access latency of NVM-buffer and NVM. The read and write latency of NVM-buffer is slightly higher than NVM, because UHS uses parallel I/O-thread scheduling to fully leverage the parallelism of NVM but introduces a little extra overhead.

TABLE II: NVM-buffer and NVM Latency

I/O	UHS Latency	NVM Latency
Write	1.74mu	1.693mu
Read	1.261mu	1.259mu

Figure 9(a) and 9(b) show the read and write latency of SSD-zone and SSD. Specially, the write latency with 4KiB I/O size of SSD-zone is much lower than SSD because of the NVM-buffer. Besides, UHS migrates the data in NVM-buffer under



(a) UHS and SSD Write (b) UHS and SSD Read

Fig. 9: UHS and SSD Access Time.

light load. When the NVM-buffer's state is clean, UHS directly reads and writes file data in the SSD. Therefore, the latency of SSD-zone is equal to SSD.

B. Macrobenchmarks

We use Filebench to perform three workloads as CreateFile, FileServer and Varmail to evaluate the overall performance of UHS. In order to make the trace match file operations better and simplify the replay, we turn on the **packed_meta_blocks** feature to place metadata sequentially during mkfs, turn off **lazy_itable_init**, **lazy_journal_init** and **has_journal** to avoid additional initialization. For a 512 GiB logical volume, the configuration of Static Address-Mapping maps block 0 to 1928764 in the volume to NVM, and maps block 1928765 to the last block in the volume to SSD. Because bcache does not support the NVM, we create a logical device using NVM and directly use mmap to read and write it. Before testing with Bcache, the size of dirty data is zero. Table III summarizes the characteristics of these three workloads.

Figure 10 shows the performance of four existing storage architectures and UHS. For metadata-intensive workloads, such as CreateFile, UHS improves performance up to **1.5x** by concurrent access to NVM and SSD. Varmail has a balance of read and write workload and FileServer has a write-intensive workload. UHS gains nearly **25%** and **53%** improvement respectively because of the NVM-buffer absorbing small write mainly. Compared with the SSD, Bcache and Device-Mapper, UHS has almost up to **8x**, **3.5x** and **6x** improvement respectively, which is mainly due to effectively exploit NVM and SSD in parallel.

TABLE III: Filebench workload Characteristic

Workloads	Avg File Size	I/O size(r/w)	R/W Ratio
CreateFile	16KiB	0/16KiB	0:1
FileServer	128KiB	1M/16KiB	1:2
Varmail	16KiB	1M/16KiB	1:1

C. Migration

When UHS is under light load or uses up a segment, the buffered data will be written to the SSD. To measure the migration overhead, we set different NVM-buffer sizes, continuously write data to NVM-buffer and record migration time of each segment.

Figure 11 reports the results. Overall, Migration time increases with the increasing size of NVM-buffer, because the migration-threads need to be waked up. We also test UHS on Polling mode where migration-threads poll to determine whether invoking migration. The execution time is slightly less. As expected, the delay of thread state switching is reduced by polling. However, UHS uses Migration Thread-Pool, setting it

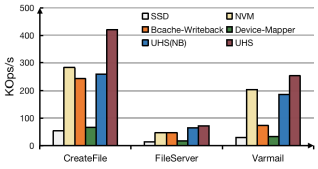


Fig. 10: Filebench

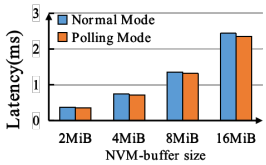


Fig. 11: Migration Time

to Poll mode consumes some CPU cycles. Therefore, we use the non-Polling model in the implementation of UHS.

V. CONCLUSION

In order to fully exploit the I/O characteristics of NVM and SSD in block-level, we propose an ultra-fast block-level hybrid storage upon NVM and SSD, called UHS. UHS presents a block-level heterogeneous storage with a unified data layout but an explicit NVM block-space, designs static and dynamic NVM-buffer accessing considering predefined storage affinity and runtime workload to fully reap the merits of NVM and SSD respectively to achieve superior performance.

ACKNOWLEDGMENT

This work was supported in part by NSFC No.62172175, Creative Research Group Project of NSFC No.61821003, National key research and development program of China under Grant 2018YFA0701800, and Key Research and Development Project of Hubei No.2022BAA042.

REFERENCES

[1] B. C. Lee, P. Zhou *et al.*, "Phase-change technology and the future of main memory," *IEEE micro*, 2010.
[2] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in Byte-Addressable persistent B+-Tree," in *FAST*, 2018.
[3] Y. Yang, Q. Cao *et al.*, "Spmfs: A scalable persistent memory file system on optane persistent memory," in *50th International Conference on Parallel Processing*, 2021.
[4] "ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory," in *FAST*, 2022.
[5] Z. Lu, Q. Cao *et al.*, "p2kvs: a portable 2-dimensional parallelizing framework to improve scalability of key-value stores on ssds," in *EuroSys*, 2022.
[6] M. Cao, S. Bhattacharya *et al.*, "Ext4: The next generation of ext2/3 filesystem," 2007.
[7] C. Hellwig, "Xfs: the big storage file system for linux," *; login:: the magazine of USENIX & SAGE*, 2009.
[8] "Supporting filesystems in persistent memory," 2022. [Online]. Available: https://lwn.net/Articles/610174/
[9] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *SOSP*, 2017.
[10] S. Zheng, M. Hoseinzadeh *et al.*, "Ziggurat: A tiered file system for non-volatile main memories and disks," in *FAST*, 2019.
[11] Intel, "Intel® optane™ dc persistent memory." 2022. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html
[12] "Fio," 2022. [Online]. Available: https://github.com/axboe/fio
[13] J. Xu and S. Swanson, "Nova: A log-structured file system for hybrid volatile/non-volatile main memories," in *FAST*, 2016.
[14] H. Song, S. Kim *et al.*, "First responder: Persistent memory simultaneously as high performance buffer cache and storage," in *ATC*, 2021.
[15] "Bcache," 2022. [Online]. Available: https://bcache.evilpiepirate.org/
[16] "device-mapper," 2022. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/index.html
[17] C. Lee, D. Sim *et al.*, "F2FS: A new file system for flash storage," 2015.
[18] "blktrace," 2022. [Online]. Available: https://github.com/sdsc/blktrace
[19] "libpmem," 2022. [Online]. Available: https://pmem.io/pmdk/libpmem/