# An Empirical Study on NVM-based Block I/O Caches

Geonhee Lee
Seoul National University
ghlee@archi.snu.ac.kr

Hyeon Gyu Lee
Seoul National University
hglee@archi.snu.ac.kr

Juwon Lee
Seoul National University
jwlee@archi.snu.ac.kr

Bryan S. Kim*
Seoul National University
bryansjkim@snu.ac.kr

Sang Lyul Min
Seoul National University
symin@snu.ac.kr

## Abstract

This paper presents an empirical study on block I/O caches when combining the performance benefits of emerging NVM storages and the cost-effectiveness of secondary storages. Current state-of-the-art I/O caching solutions are designed around the performance characteristics of SSDs, and our study reveals that using them with NVM storages does not fully reap the benefits of I/O devices with near-DRAM latencies. With fast NVM storages, locating data must be handled efficiently, but the sophisticated yet complex data structures used in existing designs impose significant overheads by substantially increasing the hit time. As this design approach is suboptimal for accessing fast I/O devices, we suggest several architectural designs to exploit the performance of NVM storages.

***Keywords*** NVM, NVDIMM, I/O Cache, Storage Stack

**ACM Reference Format:**
Geonhee Lee, Hyeon Gyu Lee, Juwon Lee, Bryan S. Kim, and Sang Lyul Min. 2018. An Empirical Study on NVM-based Block I/O Caches. In *9th Asia-Pacific Workshop on Systems (APSys '18), August 27–28, 2018, Jeju Island, Republic of Korea.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3265723.3265741

## 1 Introduction

Caching is a central design concept in today's computer systems across a number of domains. In particular, the block I/O cache located between the file system and the block device layer transparently combines two or more I/O devices to provide the performance of the faster device, with the capacity of the cost-effective one [8, 21, 26, 29]. Unlike the page cache that is tightly-coupled with the memory and file management of the operating system, the I/O cache offers a file system-agnostic implementation for improving the I/O performance. Existing implementations and research are designed around SSDs and HDDs, aiming to provide a device with flash-level latencies and a cost-competitive capacity [4, 10, 12, 19, 23, 25, 33].

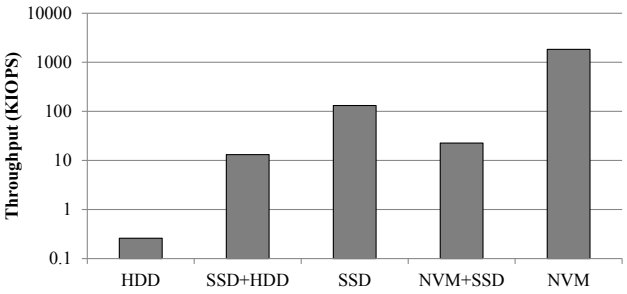



**Figure 1.** The log-scale throughput of individual devices such as HDD, SSD, and NVM, and block I/O cache device such as SSD+HDD (SSD cache backed by HDD), and NVM+SSD.

The advent of non-volatile memory-based storage (NVM storage) [5, 9, 17, 18, 27, 28], however, fundamentally changes the landscape of the storage stack [3, 11, 13, 20, 30, 32]. NVM storage offers both persistency and near-DRAM latencies, making it an attractive target as the faster device for block I/O caching. However, when using NVM as a cache, our experiments reveal that the complex management schemes of existing solutions impose significant overheads, unable to fully utilize the raw performance of the NVM.

Figure 1 illustrates this case. It shows the throughput of individual devices such as HDD, SSD, and NVM, and block I/O cache devices such as SSD cache backed by HDD (SSD+HDD) and NVM cache backed by SSD (NVM+SSD). We exercise each device with 16 threads that asynchronously and randomly read and write different files. For the SSD+HDD, the performance of the block I/O cache is better than that of the HDD, but not as high as the throughput of the SSD. For the NVM+SSD, however, this is not the case. While NVM+SSD improves over SSD+HDD, its performance is worse than using an SSD alone. This shows that using the state-of-the-art block I/O cache as-is with NVM will not improve the performance, and in fact, may degrade it.

*Corresponding author.


*APSys '18, August 27–28, 2018, Jeju Island, Republic of Korea*

ACM ISBN 978-1-4503-6006-7/18/08…$15.00
https://doi.org/10.1145/3265723.3265741

In this paper, we empirically study two block I/O caches in the Linux kernel: bcache [21] and dm-cache [29]. We evaluate these two caches across multiple dimensions such as:

- Device configuration
- Working set size
- Access concurrency
- Block I/O size
- Read/write ratio

Based on our experiment results, we suggest several architectural designs for NVM-based block I/O caches to fully exploit the performance of the fast NVM storage.

The remainder of this paper is organized as follows. § 2 gives a brief overview of I/O caching, NVM, and its long-term implications in the system software. § 3 explains the high-level architecture of bcache and dm-cache in the Linux kernel. § 4 describes the experimental setup, § 5 establishes baseline results for individual storage devices, and § 6 presents our findings for I/O caches. § 7 suggests several architectural designs for NVM-based I/O caches, and finally § 8 concludes.

## 2 Background and Related Work

In this section, we discuss several prior work related to ours. First, we briefly outline existing block I/O caching techniques, and then describe the current trend and development of non-volatile memory-based storage (NVM storage). Lastly, we cover several prior studies that redesign the software storage stack to incorporate NVM in the system memory.

Block I/O caches transparently combine multiple I/O devices to export a large storage device with improved performance, and a large body of work uses SSDs as the caching device and HDDs as the backing device. A number of implementations exists [8, 21, 26, 29] with bcache [21] and dm-cache [29] included in the Linux kernel. Other I/O cache designs focus on improving the overall hit ratio through online workload characterization [4, 12, 23] and managing SSD's internal quirks [19, 25, 33]. However, the complex data structures and management schemes designed for these purposes tradeoff cache hit time for better hit ratio. In this work, we investigate I/O caching techniques using emerging NVM devices.

Few prior studies suggest using NVM as a storage cache, but our work differs from them in the following manner. HALO [16] addresses the issue of NVM wear out by integrating the caching and wear leveling algorithm in the storage cache. However, the experimental results were obtained on a simulated environment that does not take into account of the kernel overhead, which becomes a significant factor with fast persistent storages. Bankshot [2] implements a user-level library for bypassing the kernel to access the NVM. This work, on the other hand, is vulnerable to wear as adversarial applications can wear the endurance-limited NVM.

Today, NVM storage is commonly available in two form factors: memory module and SSD. Memory module-based NVM storages are commonly known as NVDIMMs [18, 27, 28], and while some are implemented as flash memory-backed DRAM modules, those based on persistent memories are expected in the future [17]. On the other hand, Intel's Optane [9] and Samsung's Z-SSD [5] both achieve sub-10$\mu$s latency in an SSD form factor. Although Optane and Z-SSD demonstrate significant performance improvements over conventional SSDs, the NVDIMM has greater potential for disruption as it is much closer to the CPU.

This advent of fast, persistent storage challenges the traditional memory and storage hierarchy, and a wide range of system issues such as virtualization, storage tiering, and file system optimizations are actively being studied [17]. In particular, renovating the file system to use byte-addressable NVM has been an active area of research in the past decade. Some of its innovative approaches include allowing the user-level file system to log its updates in NVM [13], managing NVM in the kernel with consistency guarantees [6, 7, 31, 32], and simplifying and optimizing the journaling mechanisms [3, 14]. In this work, however, we study the performance of NVM-based block I/O devices as it can be used transparently from the file system.

## 3 Overview of Linux Block I/O Caches

Two block I/O caches—bcache and dm-cache—are part of the Linux kernel, and in this section, we describe their high-level architectures.

### 3.1 bcache

bcache maintains a copy-on-write B+ tree, and each node in the tree is log-structured. Figure 2 simplifies this complex data structure and instead shows a B-tree. The keys in the B-tree are extents (starting logical block and number of blocks), and its corresponding value is a list of pointers to the location of the data in the caching device. Keys not found in the B-tree are regarded as cache misses, and the I/O requests are then sent to the backing device. The B-tree metadata structure is maintained permanently in the caching device, and to mitigate frequent updates, bcache also maintains an in-memory global log that coalesces random updates for all the nodes.

The storage space in the caching device is divided into multiple *allocation buckets*. Allocation buckets are similar to segments of the log-structured file system [24]: they are sequentially written and only reused when all of its contents are invalid. These buckets must be garbage collected by bcache to reclaim space.

Overall, the complexity of bcache rivals to that of a small file system, and in fact, bcache is in the process of being developed into bcachefs [22]. In bcache's design, the log-structured management requires a linear search within the

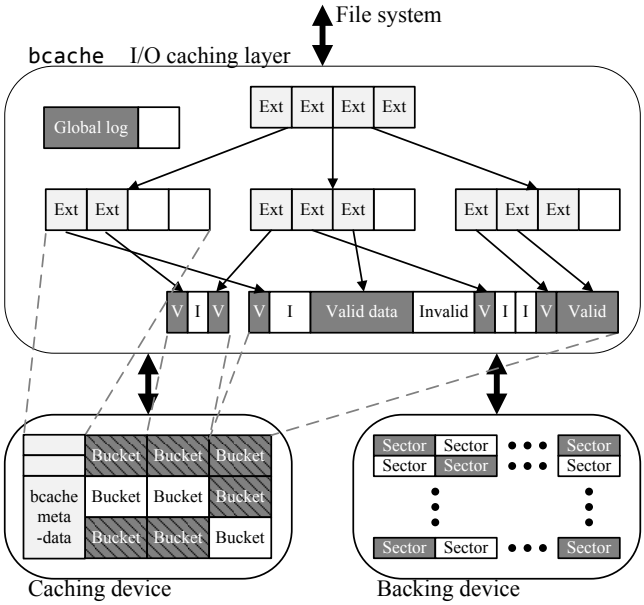



**Figure 2.** The overall architecture of bcache. The metadata structure is a variant of the B-tree, and the storage space in the caching device is managed in a log-structured manner.

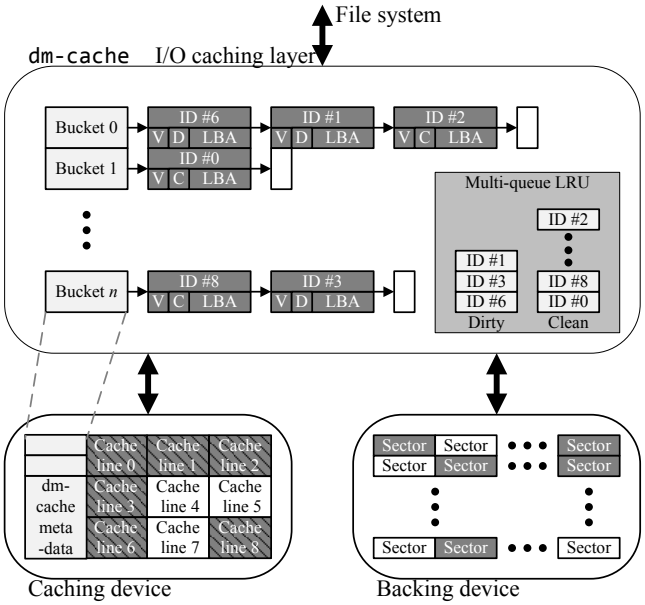



**Figure 3.** The overall architecture of dm-cache. The metadata structure is a hashed linked list with each node associated with a fixed location in the caching device.

node to locate the data, and an update to the B-tree requires locking for thread-safety. Our experimental results find these two factors as major performance overheads when accessing fast NVM storages using bcache.

### 3.2 dm-cache

Shown in Figure 3, dm-cache manages the cached data similar to that of a CPU cache. The caching device's storage space is divided into fixed-sized cache lines and data are written to the cache device in that granularity. The metadata for each cache line are kept track by a data structure that forms a hashed linked list. These data structures are made durable in a pre-allocated space in the caching device, either periodically, or upon FLUSH or FUA (force unit access) requests.

Because dm-cache caches data in a fixed size, an I/O request from the file system is first split into multiple block I/O requests aligned to the cache line size. Each smaller subrequests are then indexed in the hash table and searched through the linked list. If it is found (cache hit), the cache line associated with the node is accessed; for a miss, it must be brought into the cache from the backing device.

While bcache reclaims place through garbage collection, dm-cache must evict a cache line for new allocations. Victim for cache eviction is selected through a multi-queue clean-first algorithm; this approximates the least-recently-used (LRU) algorithm and gives preference to evicting clean data to avoid write-backs to the backing device.

The overall design of dm-cache is more simple and efficient compared to its counterpart. However, the fixed-line size is a source of inefficiency, not only for access patterns with low spatial locality, but also for large sequential accesses as they need to be chopped into smaller ones.

## 4 Experimental Setup

Our test environment consists of a 20-vCPU Intel Xeon processor on a SuperMicro X11DPi-N mainboard with 32GB of DRAM (DDR4-2666). For the block I/O devices, we use a 16GB SMART NVDIMM [27], 250GB Samsung 960 Evo NVMe SSD, and a 1TB Western Digital SATA HDD. The cost per GB ratio among NVDIMM, SSD, and HDD is 131.8:1.67:1 (based on the retail prices we spent). The system runs CentOS 7 with kernel version 3.10, and the file system is ext4 in ordered mode—only the file system metadata are journaled. All accesses bypass the file system's page cache (direct I/O mode) as the effectiveness of the page cache can be limited with NVM storages [15].

The workload is generated synthetically through fio [1], and we experiment across a diverse set of workload parameters such as: device configuration, number of threads, I/O engine (synchronous and asynchronous), block I/O size, and read/write ratio.

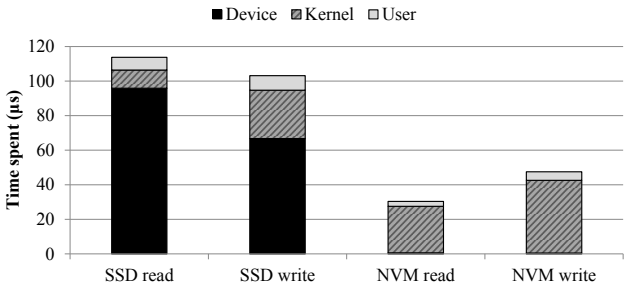



**Figure 4.** The breakdown of time spent at each layer for NVM and SSD under small random accesses.

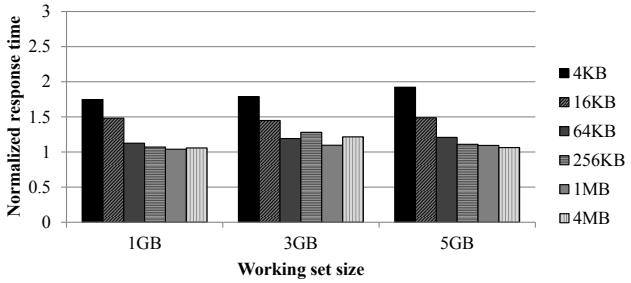
## 5 Individual Device Testing

Before exercising the I/O caches, we first measure the response time breakdown of the NVM and the SSD individually. We trace the execution of I/O with `blktrace` and post-process the traces with `blkparse` and `btt`. This gives insight to how much time is spent at the user, kernel, and device. Here, the device time includes the time spent at the I/O scheduler, the device driver, and the hardware. We use 4KB random accesses (both reads and writes) with 4 threads simultaneously dispatching requests. As shown in Figure 4, even though NVM offers much faster raw device performance, a relatively greater portion of time is spent at the kernel, mitigating the benefits of a fast persistent storage. We observe that the actual time spent in the kernel also increases with fast NVM. The kernel time includes various layers such as the VFS and the underlying file system (ext4 in this case), thus a more careful study understanding this cause is needed.

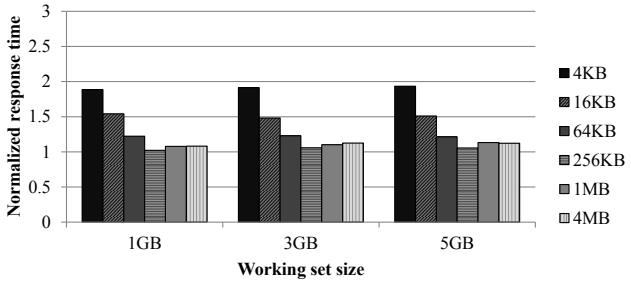
## 6 I/O Cache Evaluation Results

We measure the performance of existing I/O caching solutions, `bcache` and `dm-cache`, and empirically analyze their overheads in this section. We conduct the following experiments to isolate certain aspects of the I/O caches.

- **Lookup overhead**: Measure the response time of synchronous random reads to gauge the overheads associated with locating cached data. (§ 6.1)
- **Concurrency**: Measure the throughput of asynchronous reads with respect to increasing concurrency. We aim to understand the limitations of the internal data structures when multiple requests concurrently access them. (§ 6.2)
- **Read/write ratio**: Measure the throughput of asynchronous I/O with different write percentages. This is to understand write's effect on updating the internal data structures. (§ 6.3)
- **Working set size**: Measure the throughput of asynchronous reads as the working set size changes to observe how efficiently cache misses are handled. (§ 6.4)




**(a)** Normalized response time of `bcache`: NVM+SSD.




**(b)** Normalized response time of `dm-cache`: NVM+SSD.

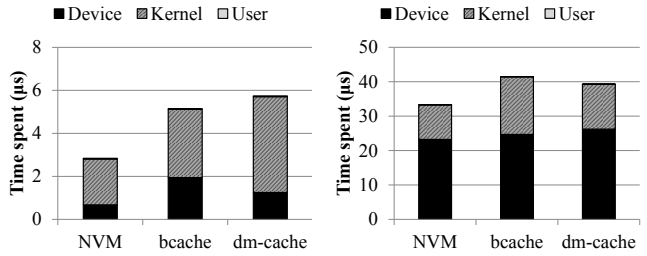
**Figure 5.** The response time of `bcache` and `dm-cache` normalized to the response time of the NVM. We measure with respect to changes in the working set size and I/O access size. Data are synchronously and randomly read by a single thread.

### 6.1 Lookup Overhead

In this subsection, we analyze the overheads associated with locating cached data. We measure the response time when data are synchronously read at various block I/O sizes and working set sizes. The working set, however, entirely fits in the caching device.

Figure 5a shows the average response time of `bcache` configured as NVM+SSD (NVM caching and SSD backing) normalized to that of the NVM when accessed without the I/O caching layer. When the I/O access size is small at 4KB, we observe that the response time of `bcache` can reach up to 1.92x of that of the NVM. This means that `bcache`'s data management overhead is as much as the combined time of the file system (but bypassing the page cache), the device driver, and the NVM device itself. For `bcache` to locate data, B-tree nodes must be traversed and searched linearly, and this is the main source of slowdown. As I/O access size increases, however, the performance degradation diminishes as more time is spent transferring data. Increasing the working set size effectively inflates the size of the B-tree in `bcache`, and in general, the overall response time deteriorates as the working set size increases as well.

The performance of `dm-cache` configured as NVM+SSD is shown in Figure 5b. The same measurement methodology from `bcache` is used to profile `dm-cache` as well. Similar to the findings in Figure 5a, the response time of `dm-cache`




**(a)** 4KB random reads. **(b)** 64KB random reads.

**Figure 6.** The breakdown of time spent at each layer for bcache and dm-cache compared to that of a raw NVM for 4KB and 64KB accesses. A single thread synchronously and randomly accessed the device.

can reach up to 1.93x of that of the NVM for small random accesses. The cause of the slowdown is traversing the linked list to locate the data in the cache. By default, dm-cache uses 256KB cache blocks, and we observe that its performance is at its best when I/O sizes match the cache block size.

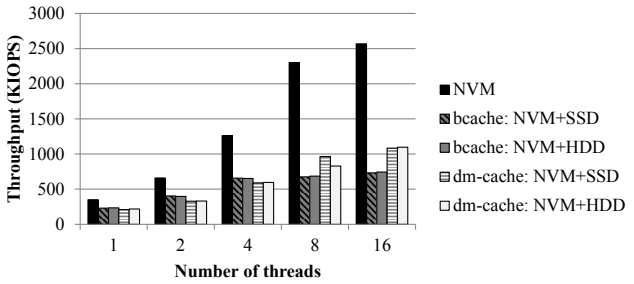
Figure 6 shows the synchronous response time breakdown of bcache and dm-cache compared to that of a raw NVM for 4KB and 64KB accesses. This reveals the difference in time spent at each layer, depending on the I/O access size. For the 4KB access in Figure 6a, the time spent at the device is relatively small, and the increased overhead of the I/O cache (exhibited by the increase in time spent in the kernel) accounts for a greater portion in the overall performance. However, for larger I/O requests shown in Figure 6b, the time spent in the device is relatively large, amortizing the I/O caching layer's overhead.

In summary, we empirically discover that the lookup overhead to locate cached data is significantly high. Both bcache and dm-cache linearly search through its data structure (log-structured B+ tree and hashed linked list, respectively), and this traversal accounts toward a non-trivial overhead when accessing fast NVM storages.

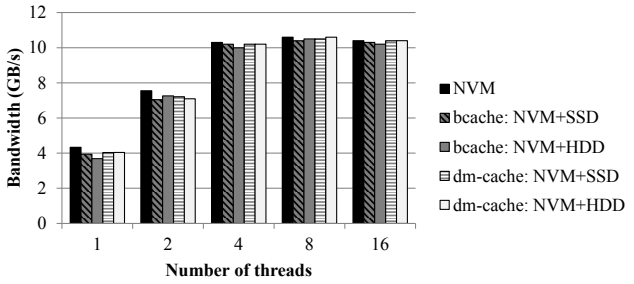
### 6.2 Concurrency

In Figure 7, we measure the performance of NVM, bcache, and dm-cache with increasing concurrency under 4KB small random reads and 4MB large sequential reads. Because the entire working set fits in the NVM storage, the performance of the backing device should have little impact.

The throughput of 4KB small random reads is shown in Figure 7a. With 1 thread, the performances of bcache and dm-cache are 67% and 62%, respectively, of that of the NVM, and become worse with 16 threads: 29% and 43%, respectively. This is because both bcache and dm-cache use locks when traversing its data structures. dm-cache locks the hash table when accessing the hashed linked list, and releases the lock once it completes its traversal. For updating the data structure, all modifications are serialized via an internal work




**(a)** The throughput of NVM, bcache, and dm-cache under small random reads.



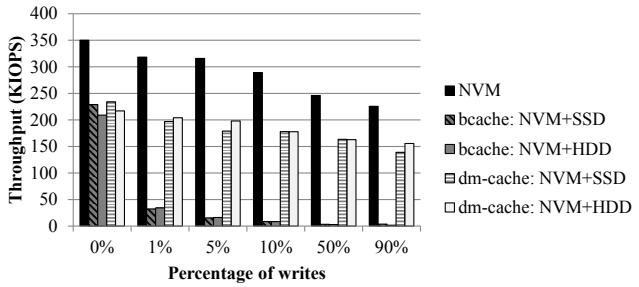

**(b)** The bandwidth of NVM, bcache, and dm-cache under large sequential reads.

**Figure 7.** The performance of NVM, bcache, and dm-cache under small random reads (Figure 7a) and large sequential reads (Figure 7b). We measure with respect to changes in the number of threads and device configuration. The entire working set fits in the NVM storage, and data are accessed asynchronously.

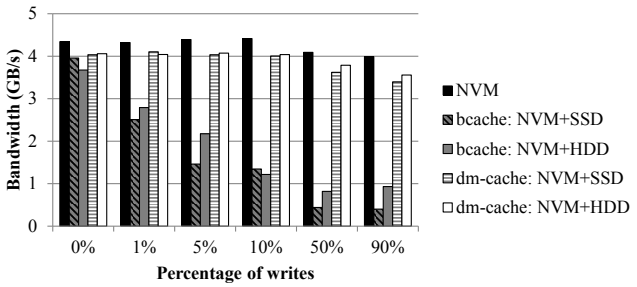
queue. On the other hand, bcache uses intention locks on all of its nodes (including root), which allows multiple reads to traverse the data structure concurrently. However, even for read workloads, because data are brought into the cache, they effectively act as writes to both the caching device and the B-tree data structures. These locks are the cause for the lack of scalability for both the caching techniques.

For 4MB sequential reads in Figure 7b, the caches' performances are as good as accessing the raw NVM device. The bandwidth saturates with 4 threads, at 10GB/s data transfer bandwidth. Although this is lower than the transfer bandwidth for a memory module (DDR4-2400 yields 19.2GB/s), this overhead is mainly due to the software storage stack (kernel and driver).

Overall, we find that the coarse-grained locking mechanisms for both bcache and dm-cache limits the scalability of these I/O caches, and NVM-conscious caches should implement a lock-free and concurrent data structures for improved scalability.




**(a)** The throughput of NVM, bcache, and dm-cache under small random accesses.




**(b)** The bandwidth of NVM, bcache, and dm-cache under large sequential accesses.

**Figure 8.** The performance of NVM, bcache, and dm-cache under small random accesses (Figure 8a) and large sequential accesses (Figure 8b). We measure with respect to changes in the percentage of writes in the workload of a single thread. The entire working set fits in the NVM storage, and data are asynchronously accessed.

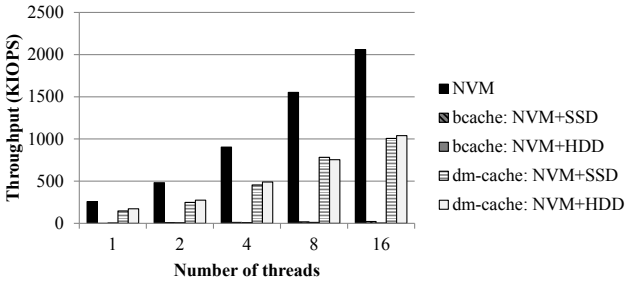



**Figure 9.** The throughput of NVM, bcache and dm-cache under 4KB random read/write (50:50 ratio).

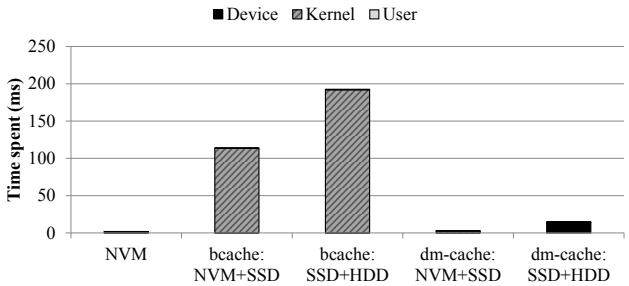



**Figure 10.** The breakdown of time spent at each layer under 4KB random read/write (50:50 ratio) for NVM and the I/O caches: bcache and dm-cache.

### 6.3 Read/Write Ratio

So far we presented performance results for read-only workloads. In this subsection, we measure the aggregate throughput and bandwidth of reads and writes for bcache and dm-cache in Figure 8. Similar to prior experiments, the entire working set fits in the NVM storage.

Figure 8a shows the throughput when a single thread reads and writes in 4KB size. When all of its accesses are reads, the results are similar to that of Figure 7a, where both bcache and dm-cache perform comparably, but not as well as accessing NVM storage directly. However, writes severely degrade the performance of bcache. Even at 1% writes, the overall throughput of bcache drops to 11% of NVM's performance, while that of dm-cache shows little change. dm-cache and directly accessing NVM also show some drop in throughput as write percentages increase, but not as drastic as bcache. We also observe this performance degradation for 4MB sequential accesses, as shown in Figure 8b. Under sequential accesses, 1% write causes the overall performance to drop to 58% for bcache. These experiments show bcache's extreme vulnerability to writes. Similarly, this observation is also made in Figure 9 when increasing the number of threads at fixed 50:50 read-to-write ratio: bcache's performance suffers when writes are added to the workload regardless the number of threads.

**Table 1.** Tabular data (in ms) for Figure 10.

| | Device | Kernel | User |
|---|---|---|---|
| NVM | 0.00074 | 1.60 | 0.0050 |
| bcache: NVM+SSD | 0.0060 | 113.53 | 0.40 |
| bcache: SSD+HDD | 0.45 | 191.52 | 0.75 |
| dm-cache: NVM+SSD | 0.0013 | 2.61 | 0.0091 |
| dm-cache: SSD+HDD | 13.89 | 0.75 | 0.023 |

Figure 10 and Table 1 show the average time spent at each layer for NVM and the I/O caches configured for both NVM+SSD and SSD+HDD. In this experiment, 8 threads concurrently and asynchronously read and write (50:50 ratio) in 4KB I/O sizes. For dm-cache with SSD+HDD, the time spent at the device occupies a significant part of the response time, amortizing the I/O caching layer's overhead. bcache, on the

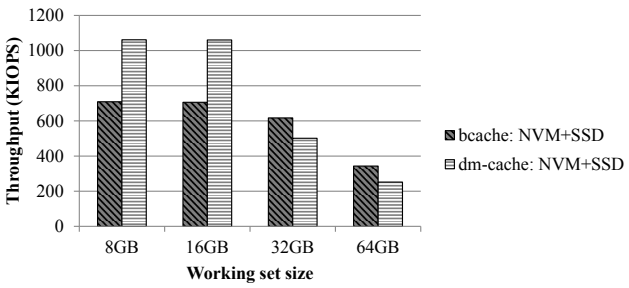



**Figure 11.** The throughput of bcache and dm-cache under small random reads.

other hand, spends significant portion of the time in the kernel, regardless of device configuration of NVM+SSD or SSD+HDD. The portion of kernel time at bcache is much larger than the results in Figure 6 because bcache's performance suffers when writes are added. This performance degradation is because bcache uses a coarse-grained lock when making changes to the B-tree data structures, and this, combined with increased fragmentation in both the caching device and the B-tree, attributes to the limitation of bcache.

### 6.4 Working Set Size

In the previous experiments, the working set size was small so that it reveals the performance of the caching layer and the NVM device. In this subsection, we experiment with various working set sizes that reach beyond the caching device's capacity. We chose read-only workload because writes degrade the performance of bcache too much and make it hard to observe performance effects due to accessing the backing device.

Figure 11 shows the throughput of bcache and dm-cache under small random reads with 16 threads. The performance of the NVM-only device cannot be measured as the working set extends beyond its capacity. As observed in Figure 7a, dm-cache outperforms bcache for small working set sizes. However, for large working sets, bcache performs better than dm-cache. Under random reads, data is evicted from the caching device as it runs out of free space. In bcache, evicting a large number of stale data is easy as the device capacity is managed in a log-structured manner: find the oldest bucket. On the other hand, dm-cache manages its space in multiple fixed-sized cache lines, and the victims must be selected according to its recency information. This makes bcache more efficient in reclaiming large space compared to dm-cache.

## 7 Design Suggestions for NVM-based Block I/O Caches

We summarize our findings and suggest designs for NVM-based block I/O caches based on our observations.

- **Finding cached data faster**: The lookup overhead to determine where the data resides adds considerable overheads in current block I/O caches. For small 4KB random reads, the response time of synchronous accesses are nearly twice as slow as accessing the NVM storage alone without caching techniques. The main overhead involved is the linear traversal of data structures. Keeping metadata sorted expedites this process, but this tradeoffs the performance of updates.
- **Fine-grained lock or lock-free data structures**: We find that locking limits the overall scalability of the block I/O caches, only able to extract a third of the NVM storage's performance. In both caches, the locks are coarse-grained, acquiring that of the root node or that of the entire hash table. Fine-grained lock, or a lock-free data structure should be considered for accessing the fast NVM storage concurrently.

## 8 Conclusion

In this paper, we measure the performance of the block I/O caches in the Linux kernel using emerging NVM storages as the caching device to the SSD or HDD backing devices. Our study reveals that current I/O caching solutions fail to fully utilize the performance of low-latency and high-throughput NVM storages, because their complex data structures make it difficult to efficiently locate data. We empirically measure and analyze these limitations, and suggest several architectural designs to benefit from fast storages. We anticipate that our findings will be a stepping-stone toward renovating the storage stack for fast storage devices.

## Acknowledgments

We thank our shepherd Cheng Li and the anonymous reviewers for their constructive and insightful comments. This work was supported in part the National Research Foundation of Korea under the BK21 Plus for Pioneers in Innovative Computing (21A20151113068) and the PF Class Heterogeneous High Performance Computer Development (NRF-2016M3C4A7952587). Institute of Computer Technology at Seoul National University provided the research facilities for this study.

## References


[1] Jens Axboe. 2018. fio. https://github.com/axboe/fio

[2] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2013. Bankshot: caching slow storage in fast non-volatile memory. In *Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*.

[3] Cheng Chen, Jun Yang, Qingsong Wei, Chundong Wang, and Mingdi Xue. 2016. Fine-grained metadata journaling on NVM. In *IEEE Symposium on Mass Storage Systems and Technologies*.

[4] Feng Chen, David A Koufaty, and Xiaodong Zhang. 2011. Hystor: making the best use of solid state drives in high performance storage systems. In *ACM International conference on Supercomputing*. 22–32.

[5] Wooseong Cheong, Chanho Yoon, Seonghoon Woo, Kyuwook Han, Daehyun Kim, Chulseung Lee, Youra Choi, Shine Kim, Dongku Kang, Geunyeong Yu, Jaehong Kim, Jaechun Park, Ki-Whan Song, Ki-Tae Park, Sangyeun Cho, Hwaseok Oh, Daniel D. G. Lee, Jin-Hyeok Choi, and Jaeheon Jeong. 2018. A flash memory controller for 15$\mu$s ultra-low-latency SSD using high-speed 3D NAND flash with 3$\mu$s read time. In *IEEE International Solid-State Circuits Conference*. 338–340.

[6] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin C. Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *ACM Symposium on Operating Systems Principles*. 133–146.

[7] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *ACM European Conference on Computer Systems*.

[8] Facebook. 2017. flashcache. https://github.com/facebookarchive/flashcache

[9] Frank T. Hady, Annie P. Foong, Bryan Veal, and Dan Williams. 2017. Platform Storage Performance With 3D XPoint Technology. *Proc. IEEE* 105, 9 (2017), 1822–1833.

[10] Taeho Kgil, David Roberts, and Trevor Mudge. 2008. Improving NAND flash based disk caches. In *IEEE International Symposium on Computer Architecture*. 327–338.

[11] Jungwon Kim, Seyong Lee, and Jeffrey S Vetter. 2017. PapyrusKV: a high-performance parallel key-value store for distributed NVM architectures. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*.

[12] Youngjae Kim, Aayush Gupta, Bhuvan Urgaonkar, Piotr Berman, and Anand Sivasubramaniam. 2011. HybridStore: A cost-efficient, high-performance storage system combining SSDs and HDDs. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. 227–236.

[13] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *ACM Symposium on Operating Systems Principles*. 460–477.

[14] Eunji Lee, Hyokyung Bahn, and Sam H. Noh. 2013. Unioning of the buffer cache and journaling layers with non-volatile memory. In *USENIX conference on File and Storage Technologies*. 73–80.

[15] Eunji Lee, Hyokyung Bahn, Seunghoon Yoo, and Sam H. Noh. 2014. Empirical Study of NVM Storage: An Operating System's Perspective and Implications. In *IEEE International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*. 405–410.

[16] Zhuo Liu, Bin Wang, Patrick Carpenter, Dong Li, Jeffrey S Vetter, and Weikuan Yu. 2012. PCM-based durable write cache for fast disk I/O. In *IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, 451–458.

[17] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2016. Non-volatile storage. *Commun. ACM* 59, 1 (2016), 56–63.

[18] Netlist. 2017. Netlist NVDIMM. http://www.netlist.com/products/vault-memory-storage/nvvault-ddr4-nvdimm/

[19] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2012. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems.. In *USENIX Conference on File and Storage Technologies*.

[20] Jiaxin Ou and Jiwu Shu. 2016. Fast and failure-consistent updates of application data in non-volatile main memory file system. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*.

[21] Kent Overstreet. 2014. bcache. https://www.kernel.org/doc/Documentation/bcache.txt

[22] Kent Overstreet. 2018. bcachefs. https://bcachefs.org

[23] Timothy Pritchett and Mithuna Thottethodi. 2010. SieveStore: a highly-selective, ensemble-level disk cache for cost-performance. In *ACM International Symposium on Computer Architecture*. 163–174.

[24] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer System* 10, 1 (1992), 26–52.

[25] Mohit Saxena, Michael M Swift, and Yiying Zhang. 2012. FlashTier: a lightweight, consistent and durable storage cache. In *ACM European Conference on Computer Systems*. 267–280.

[26] STEC. 2015. EnhancedIO. https://github.com/stec-inc/EnhanceIO

[27] SMART Modular Technologies. 2016. SMART NVDIMM. http://www.smartm.com/products/dram/NVDIMM_products.asp

[28] Micron Technology. 2018. Micron NVDIMM. https://www.micron.com/products/dram-modules/nvdimm

[29] Joe Thornber, Heinz Mauelshagen, and Mike Snitzer. 2015. dm-cache. https://www.kernel.org/doc/Documentation/device-mapper/cache.txt

[30] Qingsong Wei, Chundong Wang, Cheng Chen, Yechao Yang, Jun Yang, and Mingdi Xue. 2017. Transactional NVM cache with high performance and crash consistency. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*.

[31] XiaoJian Wu and A. L. Narasimha Reddy. 2011. SCMFS: a file system for storage class memory. In *ACM Conference on High Performance Computing Networking, Storage and Analysis*.

[32] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories.. In *USENIX Conference on File and Storage Technologies*. 323–338.

[33] Qing Yang and Jin Ren. 2011. I-CASH: Intelligently coupled array of SSD and HDD. In *IEEE International Symposium on High Performance Computer Architecture*. 278–289.