A SPDK-based User-Level SSD Cache I/O Stack

Yung-Hsiang Wang
Dept. of Electrical Engineering and Computer Science
National Chung Hsing University
Taichung, Taiwan, R.O.C.
yh.richard.wang@gmail.com

Wan-Ho Hsu
Dept. of Electrical Engineering and Computer Science
National Chung Hsing University
Taichung, Taiwan, R.O.C.
bonnie.shiu.1@gmail.com

Yung-Tung Chou
Phison Electronics Corp Research and Development Division
Miaoli, Taiwan, R.O.C.
hyam_chou@phison.com

Hsung-Pin Chang
Dept. of Computer Science and Engineering
National Chung Hsing University
Taichung, Taiwan, R.O.C.
hpchang@cs.nchu.edu.tw

***Abstract*—In recent years, utilizing solid-state drives (SSDs) as caches for mechanical hard drives has emerged as an appealing architecture for cost-effective storage solutions. Currently, the Linux SSD cache management module operates in the Linux kernel, requiring all I/O requests to the SSDs or HDDs to be trapped into the kernel and processed by the kernel I/O stack. However, lots of the kernel I/O stack components are primarily designed to access data stored on mechanical hard drives. When data has been cached in the SSDs, the involvements of these components are actually unnecessary. Furthermore, with the ultra-low latency SSDs, the overheads of system call traps and the processing timing of the kernel I/O stack are no longer negligible. To address above issues, this paper proposes a new user-level SSD cache I/O stack. In this new I/O stack, if a memory cache or SSD cache hit occurs, the data is directly accessed from the memory or SSD without trapping into the kernel, while avoiding kernel I/O stack ovherhead. Only when both caches misses occur, the requests are then passed to the kernel for further processing. The experimental results demonstrate the effectiveness of the proposed I/O stack in reducing the response time of I/O requests.**

***Index Terms*—SSD cache, page cache, SPDK, Linux, operating systems**

I. Introduction

To exploit the high access performance of solid-state drives (SSDs) and the large capacity of hard disk drives (HDDs), approaches have been proposed to extend the storage hierarchy to include SSDs as an intermediate tier between main memory and HDDs, forming a SSD-based caching storage system [1], [2]. In such a system, SSDs are treated as an HDD cache, seeking a storage system that has comparable performance to SSDs for a cost similar to HDDs. Dm-cache [3], bcache [4], and EnhanceIO [5] are all well-known SSD cache management modules in Linux kernel.

The system architecture of current I/O stack for SSD-based caching storage systems is shown in Fig.1. The top layer is the application program, followed by the operating system interface, virtual file system (VFS), page cache, file system, block I/O layer, and storage device, wherein, the SSD cache manager is located at the block I/O layer. Thus, SSDs are treated as the second-level HDD cache, where the first-level cache is the main memory.

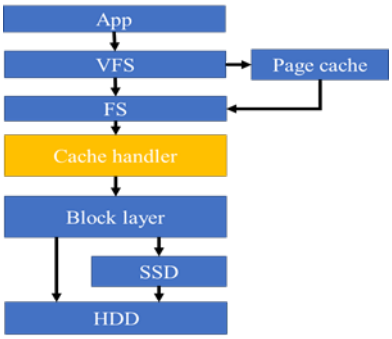

Fig. 1. The current Linux I/O stack for SSD-based caching storage systems.

As shown in Fig.1, all I/O requests that are page cache misses must go through the kernel I/O stack and according to the processing results of the SSD cache manager, reach the HDDs or SSDs. However, the VFS layer and file system exist for accessing data stored in mechanical hard disk. If the data accessed by an I/O request has been cached in the SSDs, it is meaningless for the I/O request to be processed by these components.

In other words, the current I/O stack has the following disadvantages. Firstly, all of the user requests, even if the data accessed by the requests has been cached in the memory or SSDs, must incur costly system call trap overhead. Similarly, they must pay the processing overheads of kernel I/O stack. Finally, by the current I/O stack, accessing data cached on the SSDs must go through two addressing spaces address conversions. Firstly, the file systems convert from the (file, offset) tuple, which is the file access interface in VFS, to the logical block numbers (LBNs) in HDDs. Then, the SSD cache manager must transfer from LBNs in HDDs to LBNs in SSD. And each address conversion would increase the I/O requests latency.

This problem is getting more serious with the introduction

of ultra-low latency solid-state drives (SSDs). For SATA SSDs, the access time is in the range of tens of microseconds, and the processing time of the kernel I/O stack accounts for only 8.5% [6] of the overall I/O request handling time. Therefore, the I/O processing time of the operating system kernel can be ignored. However, in recent years, the new Non-Volatile Memory Express (NVMe) [7] SSDs can provide access times less than ten microseconds. As a result, the I/O processing time of the operating system kernel becomes significant, accounting for 37.60% [6] of the total I/O processing time.

Thus, there is a need to re-examine the I/O processing flow within the operating system kernel and minimize or even remove the burden on the I/O stack. Some common approaches include using polling to avoid the overhead of interrupts and context switches [8], [9], eliminating bottom halves [10] during interrupt processing, employing simple I/O scheduling algorithms (e.g., NOOP) [11], and converting synchronous I/O handling to asynchronous processing.

In the paper, on the basis of SPDK (Storage Performance Development Kit) [12], we propose a new I/O stack for SSD-based caching storage systems. We move both the memory cache (i.e., page cache) and SSD cache managers to the user-level. In the new I/O stack, either a page cache hit or a SSD cache hit, we directly access the data from the memory or SSDs, without going through the kernel I/O stack. Only when both caches are all missed, the requests are then passed to the kernel for further processing. Since the current page cache and SSD cache managers will try to identify popular data and cache the popular data to the memory and SSD. Therefore, most of the user I/O requests will be served at the user-level and thus avoid the above-mentioned overheads, which can greatly improve the overall I/O performance.

The remainder of this paper is organized as follows. Section 2 reviews the background. Section 3 presents the new I/O stack. Section 4 provides our concluding remarks and future work.

II. Background

A. SPDK

SPDK, developed by Intel, is a user-level library that allow applications to access NVMe storage devices completely in the user-level, without going through the kernel. To achieve this, SPDK includes a user-level zero-copy and poll-driven NVMe driver; in the driver, a polling scheme is used to eliminate the I/O latency due to the kernel interrupt handling and context switching overheads. The driver also accesses NVMe I/O queues directly at the user-level, by-passing kernel during I/O operations. Thus, SPDK is considered as the I/O solution that can deliver the highest I/O performance.

B. Linux I/O Stack

To bridge the large performance gap between the processor and storage devices, page caches cache disk blocks in main memory, expecting future disk requests can be satisfied from the page cache, transferring a millisecond-based disk access into a microsecond-based memory access. Nevertheless, the capacity of a page cache is limited. Upon a page cache miss, the request must be forwarded to the underlying storage device. Besides, the page cache manager must select a victim block for replacement and write the victim block to the storage device if the victim block is dirty. Consequently, the storage device is the other shared component in the I/O path. As shown in Fig.1, in the Linux I/O stack diagram, when a process issues a disk I/O request via file I/O-based system calls, the request first arrives at the VFS layer. In Linux, VFS provides a standard interface so that user processes can uniformly access files located in different concrete file systems. Then, the VFS layer looks up the page cache to determine whether the requested data is cached in the page cache. If it has been cached in the page cache, i.e., it is a page cache hit, the cached data is copied to the user buffer and the request is completed. Otherwise, the request is passed to the file system layer.

In the VFS layer, a file is considered logically subdivided into a number of pages. Thus, reading a file is conducted in a page-based manner and the page numbers of a file are numbered continuously in the VFS layer. By contrast, the locations of blocks of data stored on storage devices are identified by logical block numbers (LBNs); a file's LBNs may not be continuous due to fragmentation. Thus, the file system is responsible for converting page-based file accesses to LBN-based I/O requests, i.e., *bio* objects in Linux.

After file system processing, the request is sent to the block I/O layer. In the block I/O layer, block-oriented I/O requests are placed in request queues, where I/O merging and scheduling are performed for possible performance optimization. Finally, the request reaches the device driver layer and is eventually issued to the storage devices to access that particular block.

III. Design and Implementation

A. Overview

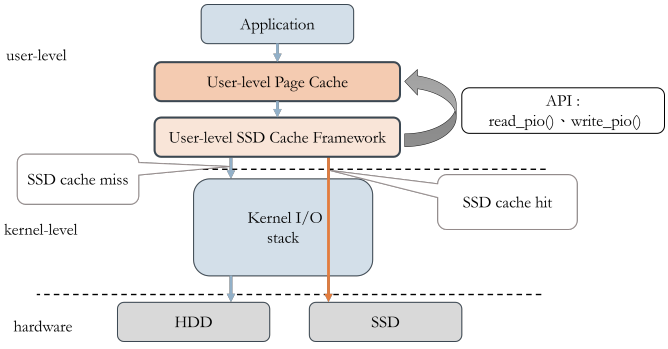

Fig. 2. The proposed system I/O stack.

The proposed system shown in Fig.2 is built on the SPDK, providing a high-performance page caching mechanism that minimizes system call overhead. SPDK offers a low-level, user-space interface for direct SSD access, eliminating kernel bottlenecks associated with traditional storage I/O operations.

In conventional architectures as shown inFig.1, applications perform I/O operations using POSIX [13] read and write

system calls, which transition into the kernel before accessing the underlying storage. This introduces context-switching and system call overhead, particularly in high-throughput scenarios. To mitigate these inefficiencies, we introduce a user-level hybrid cache system that intercepts I/O requests before they reach the kernel.

Our system provides a custom API—uread and uwrite—which applications use to interact with our page cache. Upon receiving an I/O request, the system first consults the user-level page cache. In the case of a cache miss, the request is forwarded to the user-level dm-cache, which serves as an intermediate persistent caching layer. If the data is not found in dm-cache, the request is finally routed to the SSD through SPDK for direct storage access. This hierarchical caching mechanism reduces redundant I/O operations and improves access latency while maintaining consistency.

By leveraging SPDK's user-space polling model and integrating dm-cache at the user level, our architecture minimizes kernel intervention, reduces system call overhead, and enhances overall storage performance.

B. User-level SSD Cache Mechanism

The User-level SSD Cache Framework is designed to optimize I/O performance by bypassing kernel-level overhead and directly accessing SSD storage through SPDK. Unlike traditional kernel-based caching mechanisms such as `dm-cache`, user-level SSD cache operates entirely in user space, reducing the need for system calls and avoiding the complexities of the kernel I/O stack.

1) Architecture and Core Components: The user-level SSD cache Framework consists of a user-level mapping structure, a multi-process synchronization mechanism, and a migration worker. The mapping structure is implemented using shared memory, maintaining cache metadata in a hash table where keys are derived from file paths and logical page indices. This design eliminates the need for block-level addressing, which is common in kernel-level caches.

Multi-process synchronization in user-level SSD cache is achieved using spinlocks, ensuring safe concurrent access to the mapping structure. Unlike traditional kernel spinlocks that disable interrupts to prevent race conditions, user-level SSD cache relies on atomic operations to minimize context-switching overhead. This allows multiple user processes to interact with the cache with minimal contention.

The Migration Worker operates asynchronously, handling cache promotion, demotion, and writeback tasks. When an I/O request results in a cache miss, the worker enqueues a promotion request to migrate frequently accessed data from HDD to SSD. Conversely, when cache space is limited, the worker identifies and demotes less frequently accessed data back to HDD.

2) I/O Processing Flow: When the upper page cache submits a Page I/O (PIO) request, the user-level SSD cache Framework first checks the mapping structure to determine whether the requested data resides in SSD. If the data is present (cache hit), the request is serviced directly via SPDK, leveraging zero-copy memory access to avoid unnecessary data movement between user and kernel space. In contrast, if the data is absent (cache miss), the request is forwarded to the kernel's file system, and a promotion request is issued to asynchronously migrate the data to SSD.

The framework integrates SPDK to achieve high-performance I/O operations. SPDK replaces traditional kernel-managed device drivers with a user-space NVMe driver, enabling polled-mode I/O, asynchronous event processing, and direct memory access. These features eliminate interrupt-driven context switches and significantly reduce I/O latency, particularly for ultra-low latency SSDs.

3) Comparison with Kernel-based Caching: Compared to `dm-cache`, user-level SSD cache introduces several optimizations. Kernel-based caching requires multiple layers of indirection, including system calls, block device drivers, and the device mapper framework. Each layer introduces latency and synchronization overhead. In contrast, user-level SSD cache avoids these inefficiencies by providing direct file-based mappings and bypassing kernel subsystems entirely.

Furthermore, user-level SSD cache mapping granularity is based on logical file offsets rather than fixed-size block mappings, allowing for more flexible and application-aware caching policies. The absence of kernel involvement also means that the page cache can implement customized eviction and prefetching strategies without modifying the operating system.

C. User-Level Page Cache Mechanism

To efficiently manage cached pages, our page cache employs a *free-list* and an *LRU (Least Recently Used)*. The free-list maintains available pages, while the LRU list tracks pages that have been written to, ensuring adherence to the Least Recently Used (LRU) eviction policy.

1) Write Operation: When an application calls the `uwrite` function, the system allocates a page from the free-list and copies the incoming data into it. If the data size exceeds a single page, multiple pages are allocated and linked together in a *linked list*, with their indices recorded for reference.

To prevent cache exhaustion, the system monitors the number of available pages in the free-list. If the number of free pages falls below a predefined threshold, the system evicts pages from the LRU list, writing them back to dm-cache before releasing them to the free-list. This mechanism ensures efficient memory utilization while maintaining a high cache hit rate.

2) Read Operation: When an application invokes the `uread` function, the system first checks the LRU list. The file name is hashed to obtain a pointer to the corresponding page in the LRU list. If a match is found, the page is moved to the front of the LRU list to reflect recent usage, and its data is copied to the application-provided buffer.

If the requested page is not found in the LRU list, the system invokes `PIO`, a user-level API for dm-cache, to determine if the page resides in the SSD cache. If the page is present,

it is loaded into the user-level page cache. Otherwise, in th worst-case scenario, the request is forwarded to the HDD.

By integrating a user-level page cache and dm-cach with SPDK, our system reduces kernel overhead, optimiz read/write performance, and ensures efficient cache manag ment for high-throughput applications.

IV. Experimental Study

A. Experiment Setup

To evaluate the performance of our user-level page cache d sign, we conducted a series of experiments comparing it wi the default Linux kernel page cache. All experiments we

TABLE I
System Configuration

Component	Specification
CPU	Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz
RAM	Transcend 8GB DDR4 2133 MHz
Original Device	Seagate ST500DM002 500GB
Cache Device	WDS250G3X0E-00B3N0 500GB
Operating System	Ubuntu 22.04.5 LTS
Linux Kernel	6.8.0-59-generic

performed on a machine equipped with TABLE I componer We performed a random read workload using 4KB block siz over a 1GB test file. The experiment was conducted usir `fio` with a custom I/O engine (`myfio`) built on SPDK. Th parameters we used shown in TABLE II

TABLE II
Fio Engine Setup

Parameter	Value
I/O Engine	`myfio` (SPDK-based)
Threading	Single thread
Direct I/O	Enabled (`direct=1`)
I/O Depth	1
Block Size	4KB
Total Size	1GB
Cache Size	1GB
Device	`/dev/sdc` (NVMe SSD)
Warm-up	Enabled (entire cache filled before measurement)

B. Evaluation Criteria

We focused on two primary performance metrics:

- **Input/Output Operations Per Second (IOPS):** The average I/O Operation counts in seconds.
- **Throughput:** Amount of data processed per second.

C. Experiment Result

The results highlight the performance advantage of our user-level cache system in both IOPS and bandwidth, shown in Fig.3 Fig.4. With the entire test file fitting into the user cache (1GB cache size = 1GB file size), all read operations were served directly from user space memory, avoiding any NVMe access or kernel overhead.

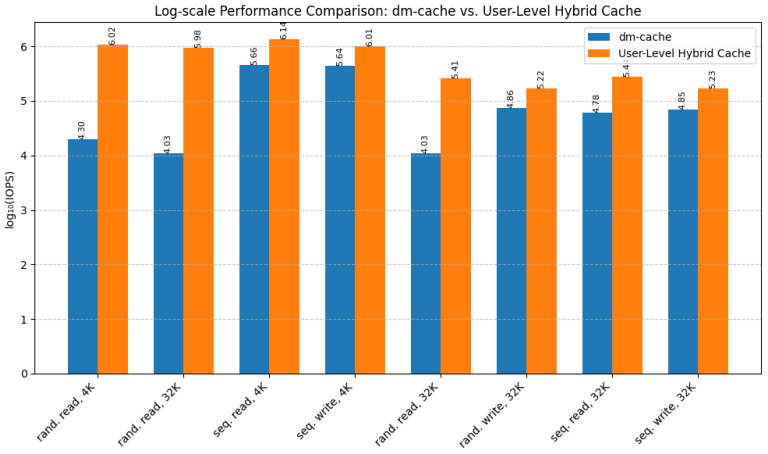

Fig. 3. The IOPS bar chart comparison

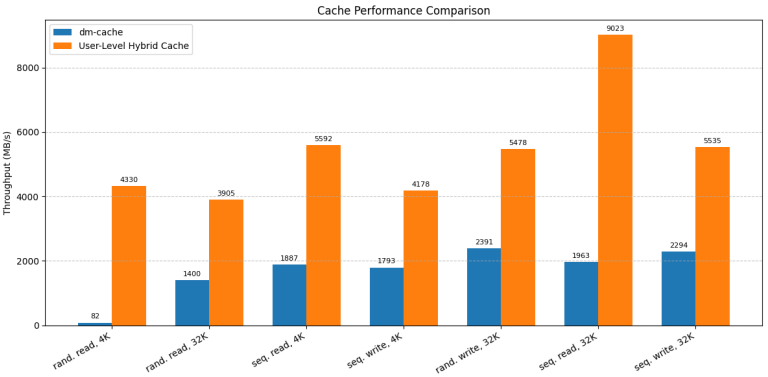

Fig. 4. The bandwidth bar chart comparison

Compared to the kernel's `dm-cache`, which involves page lookup, scheduling, and context switching, our cache leverages SPDK's polling mechanism and pre-mapped memory to deliver I/O.

This test validates the capability of our user-level caching strategy under ideal conditions where working set fits entirely in cache. It sets a theoretical upper bound on achievable read performance and confirms the efficiency of bypassing the kernel path for small block random access.

V. Conclusion and Future Work

A. Conclusion

On the basis of SPDK, this paper proposes a new I/O stack for SSD-based cache storage systems. In our new I/O stack, upon memory or SSD cache hits, the user requests are processed completely in the user-level. Only when visiting both caches are all misses, the request is then passed to the kernel, processed by the kernel I/O stack and then finally serviced by the HDDs. Since popular data are expected to be cached in the memory or SSDs, most of the user requests can be satisfied completely in the user-level, eliminating the system call trap and kernel I/O processing overhead. Thus, our work offers a

promising and practical solution for applications demanding high-performance and low-latency storage services.

B. Future Work

While the current I/O stack demonstrates the feasibility of a user-level page cache, there remain several areas for improvement and extension. First, the implementation details require further refinement to enhance performance, robustness, and maintainability. Optimizations such as more efficient page replacement strategies, better metadata management, and improved concurrency handling could significantly improve the system's effectiveness.

Moreover, this prototype was developed and tested in a constrained environment, which limits its applicability to real-world scenarios. To evaluate its full potential, the next step is to integrate the cache mechanism into a complete file system and benchmark its performance under diverse workloads and general use cases. During preliminary testing, we observed that the cache performance degrades significantly in more complex or less controlled environments.

REFERENCES

[1] R. Appuswamy, D. C. v. Moolenbroek, and A. S. Tanenbaum, "Integrating flash-based ssds into the storage stack," *Proceedings of Mass Storage Systems and Technologies Conference*, 2012.

[2] H. P. Chang, S. Y. Liao, and D. W. Chang, "Profit data caching and hybrid disk-aware completely fair queuing scheduling algorithms for hybrid disks," *Software: Practice and Experience*, vol. 45, no. 9, pp. 1229–1249, September 2015.

[3] "dm-cache," https://web.archive.org/web/20140718083340/http://visa.cs.fiu.edu/tiki/dm-cache, accessed: 2025-04-03.

[4] "bcache," https://bcache.evilpiepirate.org/, accessed: 2025-04-03.

[5] "Enhanceio," https://github.com/stec-inc/EnhanceIO, accessed: 2025-04-03.

[6] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous {I/O} stack: A low-latency kernel {I/O} stack for {Ultra-Low} latency {SSDs}," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 603–616.

[7] "Nvm express. nvm express base specification." https://nvmexpress.org/specification/nvm-express-base-specification/.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2010, pp. 385–395.

[9] J. Yang, D. B. Minturn, and F. T. Hady, "When poll is better than interrupt." in *FAST*, vol. 12, 2012, pp. 3–3.

[10] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "{OS}{I/O} path optimizations for flash solid-state drives," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 483–488.

[11] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block io: Introducing multi-queue ssd access on multi-core systems," in *Proceedings of the 6th international systems and storage conference*, 2013, pp. 1–10.

[12] "Storage performance development kit," https://spdk.io/.

[13] C. IEEE, Inc. Staff, "Information technology-portable operating system interface," 1990.