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

Yung-Hsiang Wang, Wan-Ho Hsu
Advisor: Prof. Hsung-Pin Chang

BACKGROUND

Modern storage systems demand low-latency, high-throughput I/O, especially for data-intensive workloads. Traditional Linux I/O stacks comprising the VFS, page cache, and block I/O layers introduce significant overhead due to context switches, system calls, and interrupt handling. Although the page cache improves access latency by caching frequently used data in memory, its limited size and kernel-based design reduce efficiency under heavy loads. Intel's Storage Performance Development Kit (SPDK) [1] addresses these issues by enabling user-space applications to access NVMe [2] devices directly through polling-based [3], bypassing the kernel entirely. This approach reduces latency and improves throughput, particularly in latency-sensitive scenarios. Kernel-level caching frameworks like dm-cache [4] and bcache [5] offer persistent block-level caching but still suffer from kernel overhead [6]. In contrast, user-level caching frameworks remain underexplored. Our work builds on SPDK to propose a novel user-level caching system that integrates hierarchical caching strategies with user-space execution, aiming to deliver higher performance for modern storage applications.

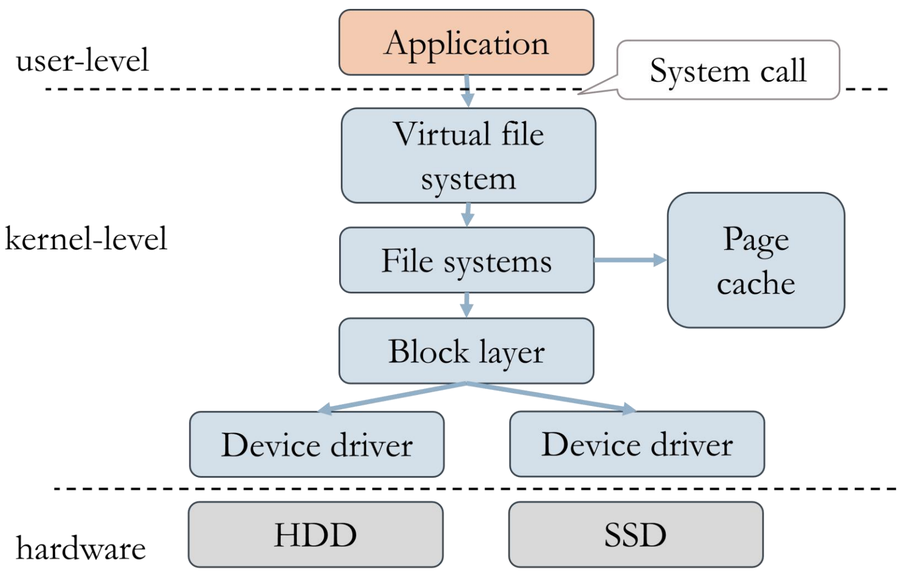

Fig. 1. The current Linux I/O stack for SSD-based caching storage systems.

DESIGN AND IMPLEMENTATION

Our system is designed to reduce I/O latency by minimizing kernel involvement in the storage stack. Built on top of SPDK, it implements a fully user-level hybrid caching mechanism composed of a page cache and an SSD cache layer, allowing applications to directly access cached data without entering the kernel I/O path. The key components include a user-level page cache, a user-space SSD cache, and custom APIs (uread, uwrite) for I/O operations.

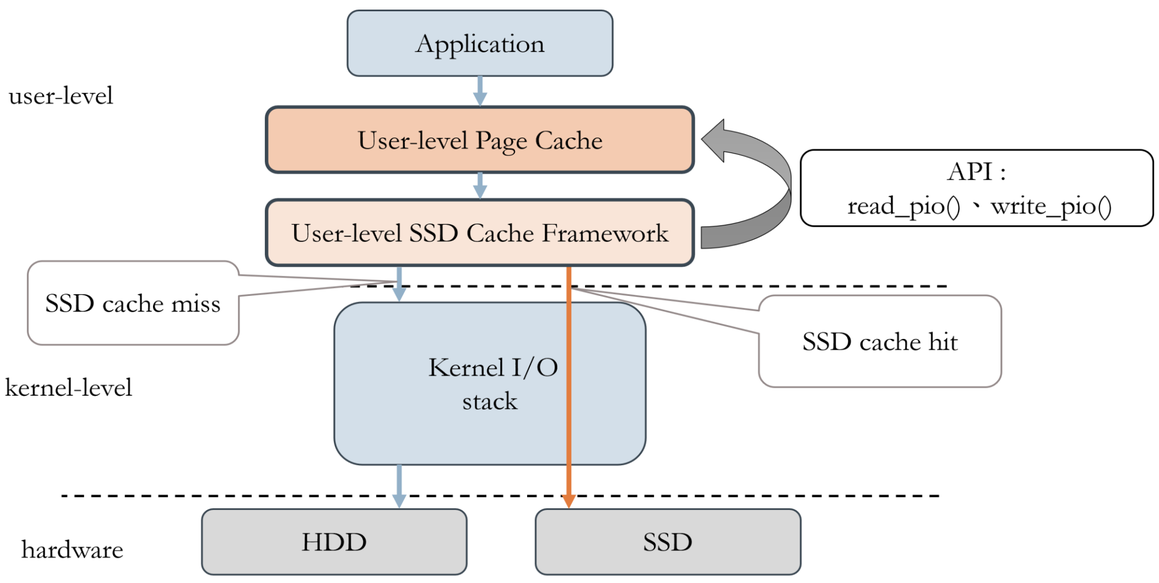

Fig. 2. The proposed system I/O stack.

The page cache manages memory pages with a free-list and LRU policy. Upon a write (uwrite), the system allocates pages from the free-list and copies data into them. If the free-list is low, dirty pages are evicted to the SSD cache via dm-cache. On a read (uread), the system first checks the page cache. On a miss, it checks the SSD cache using a user-level interface to dm-cache. Only when both caches miss does it fall back to HDD access. The SSD cache uses a shared-memory hash table to map file path and page index to cached content, supporting multiple processes via atomic operations and spinlocks. A background migration worker asynchronously promotes frequently accessed data from HDD to SSD and demotes cold data when space is low. By avoiding system calls and leveraging SPDK's polling-based NVMe access, our system eliminates context switches and interrupt handling, resulting in reduced I/O latency.

EXPERIMENTAL STUDY

To evaluate the performance of our user-level page cache design, we conducted a series of experiments comparing it with the default Linux kernel page cache. All experiments were performed on a machine equipped with TABLE I component. We performed a random read workload using 4KB block size over a 1GB test file. The experiment was conducted using fio with a custom I/O engine (myfio) built on SPDK. The parameters we used shown in TABLE II

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

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)

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. 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.

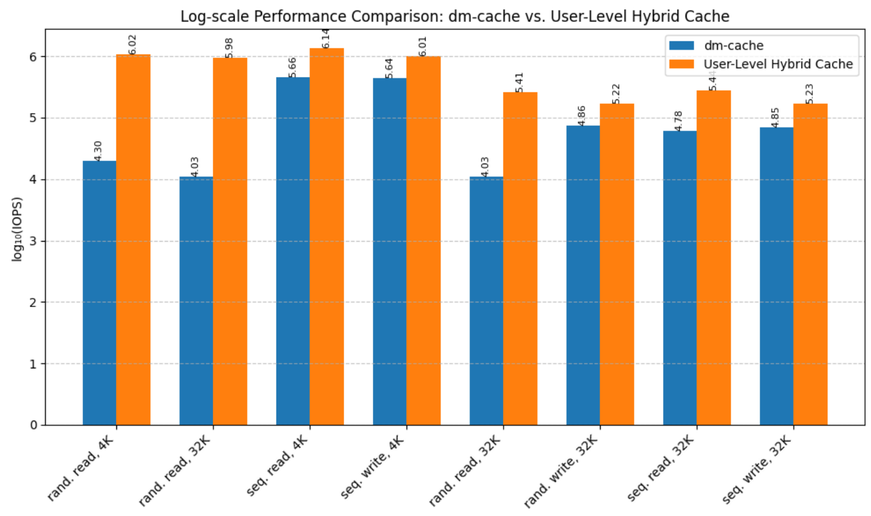

Fig. 3. The IOPS bar chart comparison

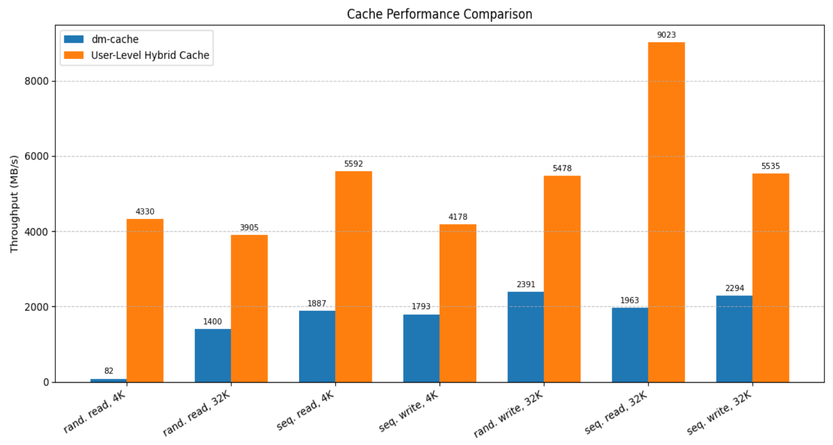

Fig. 4. The bandwidth bar chart comparison

CONCLUSION AND FUTURE WORK

We propose a user-level I/O stack based on SPDK for SSD-based caching systems. Our design avoids system calls and kernel overhead for most I/O operations. Only on full cache misses are requests passed to the kernel and HDDs. This approach significantly improves performance and reduces latency for data-intensive applications.

Future work will focus on refining implementation details, such as optimizing page replacement and metadata handling. We also plan to integrate our caching system into a full file system and evaluate its performance under real-world workloads to validate scalability and robustness.

REFERENCES

[1]"Storage performance development kit," https://spdk.io/.
[2]"Nvm express. nvm express base specification." https://nvmexpress.org/ specification/nvmexpress-base-specification/.
[3] J. Yang, D. B. Minturn, and F. T. Hady, "When poll is better than interrupt." in FAST, vol. 12, 2012, pp. 3–3.
[4] dm-cache," https://web.archive.org/web/20140718083340/http://visa.cs.fiu.edu/tiki/dm-cache, accessed: 2025-04-03.
[5] "bcache," https://bcache.evilpiepirate.org/, accessed: 2025-04-03.
[6] 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

Github Project

Our paper