OmniCache: Collaborative Caching
for Near-storage Accelerators

Jian Zhang and Yujie Ren, *Rutgers University;* Marie Nguyen, *Samsung;*
Changwoo Min, *Igalia;* Sudarsun Kannan, *Rutgers University*

https://www.usenix.org/conference/fast24/presentation/zhang-jian

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

OmniCache: Collaborative Caching for Near-storage Accelerators

Jian Zhang (Rutgers University), Yujie Ren (Rutgers University), Marie Nguyen (Samsung),
Changwoo Min (Igalia), Sudarsun Kannan (Rutgers University)

Abstract

We propose *OmniCache,* a novel caching design for near-storage accelerators that combines near-storage and host memory capabilities to accelerate I/O and data processing. First, OmniCache introduces a *"near-cache" approach*, maximizing data access to the nearest cache for I/O and processing operations. Second, OmniCache presents *collaborative caching* for concurrent I/O and data processing by using host and device caches. Third, OmniCache incorporates a *dynamic model-driven offloading* support, which actively monitors hardware and software metrics for efficient processing across host and device processors. Finally, OmniCache explores the *extensibility for newly-introduced CXL*, a memory expansion technology. OmniCache demonstrates significant performance gains of up to 3.24x for I/O workloads and 3.06x for data processing workloads.

1 Introduction

The growth in data volume and demand for high-performance data processing is driving innovative storage architectures. Traditional approaches with centralized processing and frequent data movement face performance limitations and high costs [9, 10, 44]. To address this, vendors have introduced near-storage data processing devices, bringing processing capabilities closer to storage [11, 15, 28]. These architectures leverage accelerators and host processors to enhance processing power and potentially reduce data movement and associated overheads. Realizing these benefits requires effective management and utilization of these resources. State-of-the-art near-storage designs have explored various approaches to accelerate I/O and data processing. These include using storage as a raw block device [33], developing near-storage key-value stores [13, 17, 24, 34], and creating near-storage file systems [9, 21, 31]. Additionally, application-customized techniques have been proposed [37, 39].

Utilizing memory buffers on near-storage accelerators is crucial for mitigating the impact of higher storage latency and limited bandwidth. Near-storage memory offers advantages such as localization and high bandwidth, making it an advantageous buffering medium near computational units. However, near-storage memory capacity is typically smaller than traditional host-level RAM, as demonstrated by prior studies and commercial products [11, 15]. Therefore, effective techniques that collaboratively use device and host-level memory and processors become crucial, minimizing data movement between storage and host layers, resulting in accelerated data processing and regular I/O operations (e.g., read, write).

While state-of-the-art near-storage designs improve I/O or data processing performance, they either lack any memory caching support [9, 31, 33] or fail to exploit device-level memory in collaboration with host-level memory for caching [17, 24, 44]. The absence of caching support or the failure to exploit near-storage memory for I/O and data processing increases storage access and data movement between the host and the device (e.g., fetching a 4KB block for a 1KB application request). Similarly, the absence of a collaborative host and device memory caching causes applications to stall due to cache eviction delays. Finally, prior designs use simplistic metrics to offload data processing (e.g., computing power) without considering storage-centric metrics (e.g., data distribution, I/O-to-processing ratio, data movement bandwidth, and queuing costs), leading to suboptimal performance.

To tackle the challenges above, we propose **OmniCache**, a cross-layered system software design that exploits the combined capabilities of near-storage accelerators, host CPUs, and their memory (DRAM) resources to accelerate I/O and data processing. At its heart, OmniCache introduces *a novel principle, "near-cache",* which focuses on maximizing data access on the closest cache, effectively combining the strengths of the host (such as higher memory capacity and more CPUs) and the device (being nearer to the storage) while mitigating their limitations. OmniCache employs a horizontal paradigm where application threads can concurrently store and access data from the host and the device caches, thereby improving the aggregate bandwidth and data access latency. Toward designing OmniCache, we make the following contributions:
Near-cache I/O: Firstly, we optimize I/O performance using a near-cache mechanism that simultaneously utilizes host-

level cache (*HostCache*) and device cache (*DevCache*). This near-cache approach maximizes cache utilization for various I/O access patterns, transferring only the data sizes requested by an application thread instead of the entire block from storage to the host.

Collaborative Caching for Concurrent I/O: Unlike hierarchical caching approaches where threads must wait for cache eviction to complete when a cache (e.g., HostCache) is full, OmniCache's horizontal paradigm allows threads to update the other cache (e.g., DevCache) until the eviction is complete and reduces application stalls. To locate data stored in these caches or on the disk, we introduce a scalable, host-managed indexing mechanism known as *OmniIndex*. OmniIndex utilizes a per-file interval tree equipped with a fine-grained range lock, enabling threads to access both the host and device caches concurrently for non-conflicting blocks [7].

Collaborative Processing with Dynamic Offloading: Second, we develop a dynamic offloading mechanism driven by an offloading model to accelerate data processing by leveraging HostCache and DevCache collaboratively. The mechanism enables concurrent data processing across the host and the devices and uses the caches to buffer the intermediate processing state. Beyond simple processing (e.g., data checksums and compression), we develop support for complex processing operations (e.g., K-nearest neighbor search).

Exploiting CXL.mem Capabilities: Finally, to demonstrate the adaptability of OmniCache beyond conventional NVMe-based near-storage, we exploit byte-addressable Compute Express Links (CXL) [1] with memory expansion capabilities (*CXL.mem*) to coordinate between host and device caches, reduce data movement costs and queuing delays.

End-to-end Evaluation: We evaluate OmniCache with microbenchmarks and real-world applications, including LevelDB [3] and K-Nearest Neighbor search [32]. OmniCache's near-cache I/O principle, collaborative use of DevCache and HostCache for concurrent I/O and to dynamically offload processing functions provide significant performance gains. Compared to the state-of-the-art near-storage file systems without caching [9] and those with host-only caching [44], OmniCache achieves 3.24x and 1.52x performance gains, respectively. Application write stalls are reduced by up to 2×. The collaborative approach to concurrently process data across the host and the device provides up to 3.06x gains over state-of-the-art FusionFS [9]. Finally, LevelDB [3] and data-intensive KNN [32] show up to 5.15x gains, highlighting the practical benefits of OmniCache.

2 Background and Motivation

We first present the background and related work on near-storage data processing and caching, followed by their limitations and analysis that motivates OmniCache design.

2.1 Background and Related Work

We now review prior near-storage processing studies in terms of (1) hardware trends, (2) software advancements, (3) near-storage file systems and OS support for data processing, and finally, (4) in-memory caching for storage.

Hardware Near-storage Processing Trends: Despite advancements in SSD and NVM technologies, data access and movement overheads remain dominant in I/O stacks. To address these overheads, hardware manufacturers are enhancing storage-level compute resources in near-storage processing devices like Computational Storage Devices (CSD). These devices are equipped with powerful ARM or RISC-V cores [21, 31, 34, 36, 38], FPGAs [13, 33], and significant DRAM capacity. Recent developments include CSDs with 16GB device RAM and 16-core Cortex processors [15]. They offer predefined functions and customization options to eliminate data movement between the host and the device while improving performance and flexibility [16, 38]. Moreover, CSDs such as ScaleFlux [35] and Newport [15] seamlessly integrate processor, memory, and SSD control. This integration eliminates off-chip communication and enables fast data transfer to device compute units, presenting a novel opportunity to explore the utilization of device resources.

Additionally, the emerging CXL technology (Compute Express Link) holds promise for hardware-supported memory expansion across accelerators and remote hosts [1, 19, 26]. CXL encompasses protocols such as CXL.io, CXL.cache, and CXL.mem, offering device types with different data-coherence guarantees. It enables host CPUs to expand and access device memory, with the potential to cache data on device memory for accelerating I/O and data processing.

Software Support for Near-storage Acceleration: To fully exploit the potential of near-storage accelerators, considerable software advancements have been explored to minimize data movement costs, which accelerate and efficiently leverage near-storage accelerators. Table 1 summarizes the capabilities and limitations of existing systems. Designs such as INSIDER [33] offload compute tasks to FPGA-based CSD using a block-based interface. Key-value interface designs, such as POLARDB [13], PINK [17], and KEVIN [24] offload database-specific computation to near-storage. Furthermore, NearPM [37] and SmartRec [39] focus on customized application-level optimizations or system-level guarantees.

In contrast to these systems, near-storage file system designs offload the file system closer to the storage while maintaining a POSIX-like interface. Systems such as DevFS [21] and CrossFS [31] adopt this approach by offloading metadata structures to improve performance and efficiency. FusionFS [9] compared in this work, combines file system operations with computation steps and incorporates device-level task scheduler and durability and recoverability mechanisms.

Data Processing Support: Various near-storage data processing systems have been explored. POLARDB [13] develops new application logic to accelerate applications by offloading data to FPGA-based key-value stores. λ-IO [44] utilizes an OS file system as a unified IO stack to manage computation and storage resources across the host and device.

Properties	Block-based	KV-based	Host FS	Dev-FS	Dev-FS with caching
System	Insider [33]	KV-SSD [34] PINK [17] KEVIN [24]	λ-I/O [44]	CrossFS [31] FusionFS [9]	**OmniCache**
Direct-I/O	✗	✗	✗	✓	✓
Use host Cache	✓	✓	✓	✗	✓
Use device Cache	✗	✗	✗	✗	✓
Concurrent host and device I/O processing	✗	✗	✗	✗	✓
Dynamic Offload	✗	✗	Partial	✗	✓
CXL support	✗	✗	✗	✗	✓

Table 1: Capabilities and Limitations of State-of-the-art Near-storage approaches. The last column shows our proposed OmniCache.

RocksDB	MySQL	DiskANN
99.99%	93.32%	95.89%

Table 2: Unaligned I/O Requests Ratio.

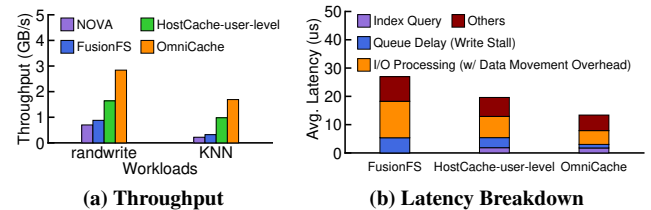

(a) Throughput (b) Latency Breakdown

Figure 1: Motivation Analysis: (a) shows aggregated throughput for an I/O intensive random write and data processing application (KNN) with 32 threads. For random write, each thread accesses a private 4GB file (128GB total); For KNN, threads share a large 128GB file with each thread accessing a non-overlapping range; (b) shows the latency breakdown for random write.

It extends eBPF for executing functions on heterogeneous hardware and provides additional programming interfaces for customized computational logic. Finally, FusionFS introduces $CISC_{Ops}$ abstraction that combines I/O and data processing operations to reduce application changes and overheads associated with I/O operations, such as system calls, data movement, communication, and other software overheads [9].

In-memory Caching for Storage: Caching I/O data in DRAM is critical for modern I/O stacks [12]. Traditional file systems rely on an OS-managed page cache, which can introduce user-to-kernel boundary crossings and substantial software overheads, often nullifying the benefits of fast storage devices [10, 21, 27, 31, 33]. While several file systems for devices like PM have disabled caching [41, 43], others are exploring user-level caching support [2, 23, 27, 30, 45]. However, *none of these storage designs have explored or designed system software for collaboratively managing on-device and host memory buffers for accelerating I/O or data processing*.

Host-level Caching for Near-Storage Processing: Caching designs like λ-IO [44] exploit host OS-level caches and only offload processing for data not in the host OS. While useful, these designs fail to consider or exploit device-level caches and suffer from the following challenges: Firstly, they lack direct I/O support, incur system call overhead, and must trap into the OS, all adding significant overheads. Secondly, they incur high data movement between the device and the host. Thirdly, host-level caching designs like λ-IO fail to concurrently support I/O and data processing across the host and the device. Fourthly, these designs lack CXL support. In contrast, OmniCache provides direct I/O bypassing the OS, reduces data movement between the device and the host, provides concurrent I/O and processing capability across the host and the device, and support for *CXL.mem* and *CXL.io*. Finally, while both OmniCache and λ-IO provide model-driven offloading mechanisms, λ-IO overlooks critical factors like device cache and command queue delays, significantly impacting performance (demonstrated in §6).

2.2 Limitations of State-of-the-art Systems

We next present the limitations of the state-of-the-art systems. Table 1 categorizes and compares current approaches with the proposed OmniCache.

Failure to Exploit Near-storage Memory for Caching: Existing near-storage designs [9, 31, 33, 44] either fail to exploit device memory or the combined capabilities of host

and device memory [42]. CrossFS [31], FusionFS [9], and Insider [33] lack support for a host or device caching, while λ-IO [44] only utilizes host caches through the OS file system. This results in high I/O overheads, data movement, and frequent kernel traps. Leveraging DevCache and HostCache together presents new opportunities to accelerate performance.

Lack of Concurrent I/O and Data Processing Support: Existing near-storage designs lack support for concurrent I/O and data processing across host and device layers [9, 31, 33]. In these designs, I/O and processing operations are mostly offloaded to the device using fewer and less powerful device-level processors and limited memory [9, 31, 33]. Host-only (OS) caching solutions like λ-IO impose concurrency limitations. The use of OS cache incurs scalability bottlenecks from coarse-grained inode-level locking and suffers from eviction stalls when the host cache is full [44]. These limitations affect the performance of I/O and data processing [9, 31, 33, 44].

Lack of Dynamic Offloading Support: Several state-of-the-art near-storage designs lack the capability to dynamically decide whether to process on the host or the device and always offload (with only partial support for λ-IO). This leads to higher data movement, queuing delays, and compute bottlenecks. In addition, all existing designs lack a holistic approach to concurrently process data across host and device layers. We show that dynamic offloading and concurrent processing can significantly enhance performance.

2.3 Analysis

First, to motivate **the importance of leveraging host and device memory for caching** and to demonstrate the significance of OmniCache toward concurrent I/O and data processing operations. We compare OmniCache against state-of-the-art NOVA (a kernel file system) [43], FusionFS (a near-storage file system without caching support), and an extended host-only caching design using OmniCache (Figure 1a). Like prior systems [9], we use a machine with 512 GB DC Optane NVM for storage, 64 CPUs, and 32 GB DRAM. We set the total cache size to 20GB for all workloads. For OmniCache, we use a 16GB host cache and a 4GB device cache. For brevity, we focus on two workloads: (1) an I/O-intensive random-write

benchmark that generates a 128 GB file with random 1KB writes using 32 threads, and (2) an I/O + processing-intensive K-nearest neighbor search (KNN) that does not fit in memory, as used in prior research [44].

For the random-write workload, NOVA exhibits inferior performance due to system call and kernel software overheads and the lack of caching support. FusionFS also performs poorly due to the absence of caching. *HostCache-user-level* suffers from data movement and frequent write stalls during cache evictions. However, OmniCache significantly improves performance by leveraging collaborative host and device cache usage (§4.2). For data-processing intensive K-nearest neighbor (KNN), *HostCache-user-level* incurs high data movement costs as it exclusively processes data on the host. In contrast, OmniCache achieves higher performance gains by concurrently utilizing HostCache and DevCache for collaborative processing (§4.3).

Second, to highlight the necessity of exploiting near-storage RAM to optimize unaligned I/O requests, Table 2 shows the substantial unaligned I/O request ratios in popular real-world applications, including RocksDB, MySQL, and DiskANN[18]. All applications exhibit exceptionally high unaligned ratios, which motivates OmniCache's collaborative I/O caching, which aims to minimize data movement overhead associated with unaligned I/O requests (§4.2). In RocksDB, we observe that for 10 million keys and a 4KB value size (fillrandom and readrandom), over 99.99% of the 141 million total I/O requests were unaligned. This emphasizes the prevalence of high unaligned I/O requests in log-structured systems. Similarly, MySQL and DiskANN (a state-of-the-art approximate nearest neighbor search algorithm) also contend with a significant number of unaligned I/O requests.

Next, to **understand the software overheads**, we show the cost breakdown of these approaches in Figure 1b. We present the average latency breakdown of OmniCache for a random write (*randwrite*) workload. Firstly, the overhead of OmniIndex is notably low, accounting for only 12% of the total time in OmniCache. Secondly, OmniCache's capability in minimizing write stalls leads to a marked decrease in the queue delay overhead. Compared to *HostCache-user-level*, the queue delay overhead is significantly reduced from 18.10% to 9.3% with OmniCache due to its ability to reduce write stalls. Furthermore, OmniCache effectively reduces data movement overhead in I/O and data processing for unaligned I/O requests owing to its efficient near-cache I/O principle.

3   Goals and Overview

Motivated by the need to exploit host and device caches collaboratively for accelerating I/O and data processing, we next discuss design goals and overview of OmniCache.

3.1   Design Goals

OmniCache introduces a novel caching mechanism to harness the potential of both host and device-level memory for caching by leveraging the hardware and software capabilities

of near-storage accelerators, host CPUs, and file systems distributed across the host and device layers. By combining their strengths and compensating for their weaknesses, OmniCache aims to achieve the following design goals:

Faster I/O using Near-cache I/O Principle: OmniCache introduces a new **near-cache I/O** principle that maximizes I/O operations to and from the nearest cache in both the host and device processors, thereby minimizing data movement. Near-cache I/O reduces data movement between the host and the device by only moving bytes actually requested by an application without requiring block-aligned data movement.

Collaborative Caching for Concurrent I/O: To address the lack of combined HostCache and DevCache use, one approach is to tier data between the caches. However, tiering hinders application threads from concurrently accessing the caches, leading to side effects like frequent CPU stalls during cache eviction. Besides, the smaller DevCache compared to HostCache complicates tiering. OmniCache addresses these challenges by supporting a horizontal paradigm that allows concurrent access to the caches. For concurrent access to non-conflicting blocks, a host-managed scalable index (OmniIndex) maps a range of blocks in a file to different caches.

Collaborative Data Processing with Distributed Caching: OmniCache exploits memory caching not just for I/O but to also accelerate data processing operations by reducing data movement between the host and the device. OmniCache achieves these goals by (1) creating mechanisms for collaborative data processing across host and device caches, (2) developing a model-driven approach to dynamically determine the optimal processing location (host or device) by leveraging hardware and software metrics (**OmniDynamic**), and (3) supporting concurrent data processing and merging results across host and device layers. OmniCache 's collaborative processing can benefit a variety of applications, especially those that involve processing and analyzing large-scale data in a parallel and distributed fashion. For example, graph processing, search engine for indexing and searching large webpage files, and NLP for extracting information and patterns from unstructured text.

Effectiveness on Byte-addressable CXLs: To minimize host-device data movement and associated queuing delays, OmniCache utilizes modern CXL technology to extend memory capacity and enable direct access to the accelerator's memory [1, 20]. Leveraging the host-managed OmniIndex, OmniCache uses CXL for direct DevCache access, thereby reducing additional data copies and queuing delays.

3.2   OmniCache Overview

We provide an overview of OmniCache components and briefly illustrate their functionalities. As shown in Figure 2, OmniCache comprises three main components: (1) a user-level library (OmniLib), (2) a user-level cache indexing structure (OmniIndex), and (3) a device manager (OmniDev). For this overview, we consider simple I/O (e.g., read(), write()) and data processing operations, e.g., read-CRC-write (read a
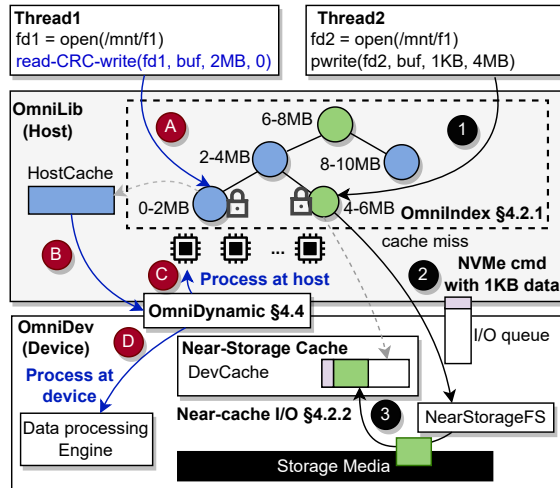
Figure 2: OmniCache High-level Design. Figure shows OmniCache concurrently handling I/O and data processing flows. For **I/O (black arrow),** ❶ application issues 1KB overwrite, which OmniLib intercepts, uses OmniIndex to locate the data in HostCache or DevCache. ❷ On a cache miss, the request is dispatched as an NVMe command using an I/O queue. ❸ OmniDev fetches the request, reads a 4KB block from storage to DevCache and updates the block. For **data processing operation (blue arrow),** Ⓐ application invokes OmniLib' read-CRC-write, which searches OmniIndex Ⓑ, uses a dynamic model to process in the host (Ⓒ) or the device (Ⓓ) or collaboratively on both.

4KB data block, calculate the checksum, and write it back).

OmniLib: The user-level component exploits host-level multicore CPUs, enables collaborative caching, and performs data processing. OmniLib performs various tasks such as dispatching I/O requests, managing cache resources, facilitating concurrent I/O and data processing operations, and handling data evictions.

Figure 2 illustrates the flow of operations across OmniCache 's components. When an application opens a file and initiates an I/O operation, OmniLib intercepts the system calls and creates a per-file I/O queue. Next, OmniLib converts all POSIX I/O calls to NVMe-like commands and adds them to the I/O queues for device (OmniDev) processing (shown from ❶ to ❸). OmniLib also handles data processing operations and offers predefined application interfaces such as *read-checksum-write* (Ⓐ) or *read-compress-write*. These operations are converted into a vector of NVMe commands and added to the I/O queue for either offloading to the device or processing at the host.

For caching, OmniCache divides the responsibilities across the host and the device layers. OmniLib provides the indexing for the HostCache and the DevCache, checks the presence of data using the index, and manages HostCache. OmniLib also decides when to evict from HostCache and DevCache by implementing a two-step LRU eviction.

For data processing, OmniLib implements a model-driven offloading engine to dynamically (Ⓑ) decide whether to offload processing (e.g., read-CRC-write) to the host (Ⓒ) or the device (Ⓓ). Finally, OmniLib also provides extensibility

to use *CXL.mem* by directly copying data and commands to DevCache and avoiding queuing delays and data copies.

Fine-grained Indexing (OmniIndex): We implemented it as a part of OmniLib, providing scalable indexing for efficient data retrieval. OmniIndex locates the data stored in HostCache, DevCache, or storage. In addition to collaborative and concurrent use of the caches, it performs ownership management of block ranges in a file and data eviction. Figure 2 shows OmniIndex represented by a per-file range tree indexing structure. Each node corresponds to a specific range/segment of a file, with a pointer to the memory buffer in the HostCache or the DevCache, or the storage. Blue and green nodes in the figure indicate data residing in HostCache and DevCache, respectively. OmniIndex's fine-grained range locks handle concurrent I/O and processing requests across threads, ensuring conflict-free access across the host and the device. OmniIndex also tracks dirty data for cache eviction (§4.2.2).

OmniDev: The near-storage component consists of a file system, a data processing engine, and support for near-storage caching. OmniDev's file system is similar to the prior near-storage file system, comprising in-memory and on-disk metadata structures and journaling for crash consistency [9]. The data processing engine handles processing requests, retrieving them from the I/O queues, updating NVMe commands with the processed output, and setting a completion flag.

With respect to caching, OmniDev handles the allocation (space management) of the DevCache using a simple device-level memory manager. On a cache miss for an I/O or processing request, OmniDev allocates space within the DevCache using its internal memory allocator, processes the request, and returns the allocated cache block's address (a block number, see §5) to update the OmniIndex using OmniLib. The coordinated cache management between the OmniLib (host-level) and OmniDev (devel-level) provides efficient cache management and fast data lookup.

4 Design

We describe OmniCache's architecture, followed by its scalable approach to use and manage HostCache and DevCache for I/O and data processing operations.

4.1 Cache Architecture

OmniCache aims to minimize data movement and software overhead by performing I/O and data processing closer to the data, resulting in higher IOPS. It utilizes both host and device caches. We discuss the rationale for distributed cache management's placement, management, and challenges.

Host Caching in the Userspace: We implement HostCache allocation and management in the user-level OmniLib to avoid kernel traps and system call bottlenecks of an OS-level cache and customize cache admission and evictions for I/O and data processing. Each application is reserved with a configurable cache memory, which is managed by OmniLib. To support file sharing and the host (and device cache) across applications, OmniCache implements cache as a shared memory (§4.2.2).

Device-level Cache: Two important design considerations for DevCache are: (1) maintaining data exclusively vs. inclusively in the host and the device cache without increasing data movement and communication overheads; (2) ensuring that applications only with correct permissions access the cache blocks despite the direct I/O bypassing the OS. Regarding (1), *we employ an exclusive caching approach,* where data blocks are either stored in the HostCache or the DevCache or the storage (which results in a cache miss). We use exclusive instead of inclusive cache (where data could be duplicated in the host and device caches) for the following reasons: Firstly, exclusive cache avoids duplicate blocks across caches, unlike inclusive cache, increasing cache coverage. Secondly, an inclusive cache to maintain consistency can incur high communication costs between the host and the device. Finally, because of the significantly different HostCache (larger) and the DevCache (smaller) capacities, an exclusive cache provides the flexibility to vary the eviction frequencies.

Regarding (2), the OS and the OmniDev manages the permission checks and access control, which we inherit from prior systems [9, 31, 33]. Briefly, for a process to access a file, a per-file I/O queue is only created by the OS if the process has access permission to the file and the queue is also tagged with the credential by the OS. OmniDev, before dequeuing and dispatching a request, checks if the request in the I/O queue has the necessary permission to access/update the file's content before checking the DevCache and the disk.

4.2 Collaborative Caching for I/O

We first discuss the techniques employed by OmniIndex for fast indexing and locating cache blocks. We then elaborate on how OmniCache reduces data movement during various I/O operations by adhering to the near-cache I/O principle. Finally, we describe how collaborative caching enhances concurrent I/O by mitigating eviction stalls.

4.2.1 Scalable OmniIndex

We address the above challenges by designing OmniIndex, a scalable and highly concurrent cache indexing mechanism based on a range tree. OmniIndex indexes data in both Host-Cache and DevCache, and is managed only by the host (OmniLib). Managing OmniIndex exclusively in the host avoids communication and consistency overheads of maintaining OmniIndex between the host and the device and leverages multicore CPU parallelism for concurrent index lookup. It also provides the flexibility to customize OmniIndex and use it for cache admission and eviction based on the application and user requirements.

OmniIndex, a per file range tree, offers a unified view of the host and device caches and the storage. Each OmniIndex node represents a specific data range within the file, with blocks potentially residing in the HostCache, DevCache, or the storage. In Figure 2, blue-colored nodes indicate data in the Host-Cache, green nodes represent blocks in the DevCache, and all others represent blocks on the storage device. OmniCache utilizes OmniIndex to determine the data location.

The I/O or data processing requests are assigned to the device by the host, which uses the OmniIndex to locate existing data blocks, if not present in any caches, allocate and updates the OmniIndex with a new node. The device CPUs do not access or update the OmniIndex.

Concurrent Non-Conflicting Access: For concurrent access/updates to a file's non-conflicting data blocks by the host CPU threads, each node range in the OmniIndex is equipped with a read-write lock. Threads acquire per-range lock before accessing the corresponding data from the cache, performing I/O, or processing. When the data is not in the host and the device caches, a range lock is acquired before issuing an I/O command. We shortly discuss the details of using the OmniIndex to perform I/O (e.g., write, read) and processing.

Avoiding Conflicts: To prevent conflicting and concurrent updates of range by the host and the device CPUs, OmniCache employs a range-level ownership model, by assigning an ownership of a range to the host or the device. This is feasible because the host OmniLib is responsible for offloading I/O or processing requests to the device and ensures that only one entity (host or device) can modify the data within the node range at any given time and preserve data integrity.

Tracking Dirty Data for Persistence: OmniIndex is essential for managing data in the host and device. Each OmniIndex node includes a dirty bit for each range and a bitmap array to track block dirtiness within a range. To update the HostCache or DevCache, pages are allocated, and the OmniIndex is updated at the range level by setting a block's dirty bit in the range. Dirty bits are set for updates and cleared during file commits or flushes (e.g., `fsync`).

Memory Overheads of OmniIndex: The memory overhead of OmniIndex is minimal. For a 1TB file, regardless of the data location (host or device cache), the index only needs 128MB (< 0.001%) of memory, with each OmniIndex node occupying 256 bytes. To reduce memory requirements further, one could have larger OmniIndex ranges or employ huge pages, which we will focus in the future.

4.2.2 I/O Operations with OmniCache

We next discuss a basic set of I/O operations, such as write and read, when using OmniCache, which mainly aims to perform near-cache I/O to avoid data movement.

Write: When an application performs write operations to expand a file, OmniCache employs the near-cache I/O principle. Initially, the data is written to the HostCache of a OmniIndex node, followed by updating the node's information, including updating OmniIndex node's dirty bit information. Furthermore, to reduce the depth of the OmniIndex and enable batch eviction, writes are merged into a single node with a maximum range of a pre-configured size (default is 2 MB). However, cache pages are allocated at the granularity of the block size. We will shortly discuss the concurrent and collaborative approach to update HostCache and DevCache.

Overwrite: To optimize overwrites, OmniCache follows a near-cache I/O approach to minimize data movement over-

head. If the blocks to be overwritten are already present in the HostCache or DevCache, OmniCache updates the cache and marks the corresponding range as dirty. In case of a cache miss, unlike existing designs, OmniCache avoids fetching the entire range of blocks from storage to the host. Instead, it only reads the relevant block(s) to the DevCache to reduce write amplification between the device and the host, applying changes directly to the blocks in DevCache and updating the OmniIndex. For instance, consider a scenario where an application issues a 1KB `write()` on a 4KB block not present in the cache. Other recent system designs must perform block-aligned reads from storage to the host, reading the entire 4KB block, resulting in I/O amplification and data movement cost. However, OmniCache leverages the benefits of the DevCache to overcome this limitation. The advantages of using Dev-Cache are further demonstrated in §6.

Read: OmniCache first searches OmniIndex to locate the blocks, then reads the blocks from HostCache, DevCache, storage, or all. Read operations work like overwrites by loading the missing data block to DevCache if space is available, and only returning requested data to the application. Similarly to Linux, OmniCache identifies access patterns to enhance prefetch granularity (up to 2MB). Importantly, for blocks in multiple OmniIndex nodes or storage, the OmniIndex with fine-grained lock is used to concurrently read blocks, resulting in lower latency and higher throughput.

File Commit (fsync): OmniCache uses OmniIndex's range and per-block dirty bit to commit one or more blocks to storage. For blocks in DevCache, OmniCache creates and issues an I/O command, and OmniDev handles the file commit. The `fsync` is treated as a barrier operation on a file.

File Sharing: For sharing files across processes, Omni-Cache allocates cache pages within a shared memory region facilitated by our shared memory allocator. Access to the shared cache and OmniIndex is limited to processes with the necessary file permissions. In the case of processes with write permissions, OmniCache maps the shared memory as writable. However, like many prior user-level direct-access file system designs, the direct-I/O approach is susceptible to corruption [9, 21, 24, 25, 29, 33]. For instance, a process with write permission could potentially corrupt the OmniIndex. While prior near-storage designs [31] transition to the OS to handle shared file updates, we have also adopted an approach used by previous user-level file systems (such as Aerie [40], Strata [25], uFS [27]), which involves a trusted third-party server mediating access to the shared OmniIndex using lease-based locks. Nevertheless, our future work will optimize inter-process sharing, as this work primarily focuses on accelerating single multi-threaded applications.

4.2.3 Concurrent Caching and Reducing Eviction Stalls

Concurrent use of HostCache and DevCache and low-overhead cache eviction is crucial for minimizing application stalls and optimizing performance. However, the limited capacities of caches can result in frequent evictions for data-intensive applications, affecting performance [22].

To tackle these challenges, we propose collaborative caching and concurrent eviction. In collaborative caching, application threads concurrently use HostCache and DevCache, switching between them when one cache is full. This reduces compute stalls and enhances performance (see Section 6).

Two-step LRU Eviction: The effectiveness of dual-cache utilization depends on accelerated cache eviction. New data updates initially enter HostCache. When HostCache reaches capacity (no available space), OmniLib directs writes to Dev-Cache and starts concurrent eviction in HostCache. A background eviction thread manages two levels of LRU information: (1) a file-level LRU, where all closed or inactive files are added to a global LRU list; (2) a per-file OmniIndex LRU list that tracks least-recently-used ranges. A file or a range becomes LRU if not accessed within a configurable 30-second epoch, akin to Linux. In the first step of the eviction process (from HostCache or DevCache), we evict LRU files. If free cache space drops to a lower threshold (10% free memory), applications continue inserting into the cache. Otherwise, the second step entails range-level LRU eviction, removing blocks from the device and host. For DevCache eviction, OmniLib sends an NVMe-like eviction command to OmniDev to evict ranges. The collaborative caching and two-step eviction ensure a seamless transition between HostCache and DevCache, concurrent utilization of different caches, reduced compute stalls, and improved performance (see Section 6).

4.3 Collaborative Processing with Caching

OmniCache also leverages HostCache and DevCache to accelerate data processing. The effectiveness of offloading operations for near-storage processing depends on the frequency of I/O operations and factors such as the impact of using less powerful on-device computing and smaller memory resources. We first discuss the application interface support, the challenges, and solutions to support collaborative processing.

Application Interface for Processing: Like prior approaches, OmniCache requires applications to use pre-defined processing functions [9, 33, 44] provided by OmniLib. Unlike λ-IO, which utilizes eBPF, imposing restrictions on operations/functions involving floating-point calculations and unbounded loops. OmniCache follows a near-storage file system paradigm; it borrows and extends the CISC-like interface from prior work [9], enabling application developers to offload richer operations/functions, such as checksum or compression. As shown in Figure 2, the I/O and data processing operations are converted and stacked as a vector of NVMe commands and offloaded as batched operations (referred to as CISC operation [9]). Unlike prior systems, OmniCache can collaboratively process simple and complex operations (e.g., *read-cal_distance-nearestK*) in the host, the device, or both.

Challenges: Collaborative data processing requires concurrent host-device processing, dynamic data migration across caches, and intelligent decisions on offloading based on hardware and software costs (e.g., data movement overheads, pro-
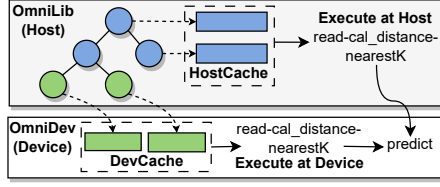
Figure 3: OmniCache Collaborative Processing for KNN. *read-cal_distance-nearestK* is concurrently executed across host and device.

cessing times, and queuing latency).

Key Ideas: To address these challenges, we extend collaborative caching approach, incorporating I/O caching and data processing. Firstly, we use OmniIndex for concurrent range-level data processing on both host and device with fine-grained range concurrency, enhancing performance. Secondly, we introduce a dynamic model that considers hardware and software metrics (e.g., storage, memory, compute time, queuing latency). This model continuously monitors the system and dynamically selects the best offloading location to use resources efficiently.

4.3.1 Extending OmniIndex for Compute Cache:

We improve OmniIndex by adding *processing buffer*, an intermediate computation buffer separate from cache buffer. Processing buffer is linked to each tree node and can be stored in host or device memory. Processing buffer is accessed via an address reference in each interval tree node. This helps OmniCache quickly find processing states, split and merge processing on the host and the device.

Case study: K-Nearest Neighbor Search (KNN): We demonstrate KNN, a widely used ML algorithm that identifies the K-nearest data points to a given query point. Since the dataset exceeds the memory capacity, KNN reads a large data chunk, calculates distances between data points, selects the K-nearest points, predicts classification based on them, and optionally writes results to a new file.

In OmniCache (see Figure 3), we handle this with a combined I/O and data processing operation called `read-cal_-distance-nearestK`. This operation reads a data range, computes distances, and selects the K-nearest points. Intermediate results, like calculated distances, are stored in a processing buffer. OmniCache executes `read-cal_-distance-nearestK` concurrently on both host and device, leveraging collaborative processing.

Afterward, OmniCache merges the K-nearest points for final classification prediction. It involves transferring data between host and device, requiring data copying. Importantly, `read-cal_distance-nearestK` and prediction operations can be performed on either host or device, determined by our resource-driven dynamic offloading strategy (§4.4).

4.4 Resource-driven Dynamic Offloading

OmniCache aims to increase near-cache and near-data processing on both host and device while minimizing data movement. However, disparate compute capabilities, cache capacity, and data transfer times between storage, HostCache, and DevCache necessitate a dynamic approach for determining

the optimal processing location. The **challenges** in making these offloading decisions are threefold. First, processing operations where the data is distributed across host and device caches may incur data movement. Second, hardware and software metrics to decide where to offload, such as computational costs, memory requirements, and I/O frequency, can vary significantly based on the processing complexity. Third, monitoring both host and device hardware and software metrics without interfering with data-plane operations is critical.

Model-driven Approach: To address these challenges, we introduce **OmniDynamic**. It leverages a model-driven approach coupled with ongoing monitoring of both host and device resources. We begin by outlining the model and then detail our implementation approach. The model (Equation 1) estimates the processing time for each request and determines where the request will be processed (host or device).

T_h and T_d calculate the processing time for a request on the host and the device, respectively, by considering various factors: **Data Ratio (R)** represents data associated with a request distributed across HostCache, DevCache, and storage. The ratios R_{hm}, R_{dm}, and R_s represent the portion of data in the host memory (*hm*), device memory (*dm*), and storage (*s*) for each request. **Execution Time (E)** captures the processing cost alone, Eh_{avg} represents the average time to execute a request on the host, while Ed_{avg} represents the average time on the device. **Data Transfer Cost (B)** captures the data movement between HostCache, DevCache, and storage. B_{hm_dm} denotes the data transfer bandwidth between HostCache and DevCache, B_{ds_hm} represents the bandwidth between storage and HostCache, and B_{ds_dm} represents the bandwidth between storage and DevCache. Finally, **Queue Latency** represents the completion time of a request, which depends on the queuing delay. This varies based on the number of I/O and data-processing requests in the per-file I/O queue and the average time required to process a request ($Cmd_{avg} * Q_{len}$). The queue delay increases during cache eviction.

$$T_h = R_d D/B_{hm\_dm} + R_s D/B_{ds\_hm} + Eh_{avg}$$
$$T_d = R_h D/B_{hm\_dm} + R_s D/B_{ds\_dm} + Cmd_{avg} * Q_{len} + Ed_{avg}$$
$$(1)$$

Continuous and Low-interference Monitoring: To realize the dynamic offloading model, OmniCache continuously monitors hardware and software parameters across the host and device layers using the OmniDynamic component. This component operates at the intersection of OmniLib and OmniDev, collecting metrics such as cache data ratios, data movement bandwidths, processing costs, and queue wait-time overheads.

For metric collection at the device, we extend each NVMe command for data processing with additional counters, including on-device processing time (Ed), data movement bandwidth between DevCache and HostCache (B_{hm_dm}), and between storage and DevCache (B_{ds_dm}).

As depicted in Algorithm 1, the initial phase is devoid of resource parameters and relies on OmniIndex to manage data distribution across HostCache, DevCache, and storage.

Algorithm 1: Model-driven Data Processing

```
1  All measurements are done periodically (per epoch)
2  Query OmniIndex to get data distribution ratio (R_h and R_d)
3  Get current queue length from I/O queue (Q_len)
4  Compute T_h and T_d based on Equation 1
5  if T_h <= T_d then
6  |    load_data_to_host()
7  |    measure_devmem_to_host_bw(&B_hm_dm)
8  |    measure_devstorage_to_host_bw(&B_ds_hm)
9  |    execute_request_at_host()
10 |    measure_avg_execution_latency(&Eh_avg)
11 else
12 |    move_data_to_device()
13 |    measure_host_to_devmem_bw(&B_hm_dm)
14 |    measure_devstorage_to_devmem_bw(&B_ds_dm)
15 |    execute_request_at_device()
16 |    measure_avg_execution_latency(&Ed_avg)
17 |    measure_queue_latency(&Cmd_avg)
```

By leveraging the cache ratios (R), OmniDynamic makes offloading decisions with a preference for near-data processing. When data exclusively resides on the host or the device, it undergoes processing without requiring data movement. Conversely, data scattered across caches and storage prompts data transfer, typically from the layer with a smaller data footprint.

OmniDynamic calculates the average processing times (Eh_{avg} and Ed_{avg}) for each request type by tracking the processing cost in the host or device. To measure the time spent by requests on the per-inode I/O queue ($Cmd_{avg} * Q_{len}$), OmniLib updates the queue admission time, while OmniDev updates the request completion time in the per-inode I/O queue.

4.5 Exploring CXL Extensibility with OmniCXL

To explore OmniCache's potential with emerging technologies like CXL.mem for caching, we introduce OmniCXL. As explained in §2.1, CXL.mem enables direct host access to an accelerator's memory. We do not assume hardware-supported cache coherence between HostCache and DevCache. In this context, we investigate how OmniCache leverages CXL.mem to reduce data copy and queuing bottlenecks while ensuring safe operation without hardware-level cache coherence support. In OmniCache's queue-based design, all requests incur overheads like packing and copying data and NVMe commands from the application (or device) buffer to the DMA-enabled I/O queues, queuing delays, and host CPU overheads like polling for request completion.

Reducing I/O and Processing Overheads: To reduce the above overheads, we propose extending OmniCache to leverage *CXL.mem* (OmniCXL). In this approach, device memory appears as an additional NUMA node in the host OS, a widely used abstraction for memory expansion. To use the device memory as DevCache, the OmniLib of a process memory maps and registers a designated region within the address space as DevCache. The DevCache size for each application is determined based on a specified limit.

When an application issues I/O operations like `write()` that cannot be cached in the HostCache due to space constraints, OmniLib directly writes data to DevCache after ac-quiring the range lock in OmniIndex and flushes the data to memory. This enables OmniCXL to avoid (1) copying data between application and device queues, (2) packing (at host) and unpacking (at device) NVMe commands, (3) reducing queuing delays, and (4) continuous polling for request completion, thereby minimizing CPU overheads. Furthermore, data copy overheads are avoided for processing requests offloaded to the device (e.g., `append-checksum-write`), but the request queuing and polling are still necessary.

5 Implementation Details

We first describe the implementation details at the near-device and host layers, followed by our approach to emulate near-storage and CXL.mem.

First, we implement an emulated **OmniDev** as a device driver (8K LOC) due to the lack of access to a programmable storage hardware, similar to prior work [9, 31]. To understand OmniCache's impact on faster and slower storage, we implement two distinct near-storage backends: one on Intel Optane Persistent Memory (PM) by extending PM file system and the other on NVMe-based SSDs that uses block-level ext4 with I/O operations bypassing the OS cache. We also add a storage processing engine to OmniDev.

Second, to manage DevCache, in OmniDev, we implement a lightweight and efficient memory allocator that uses a bitmap array to track the availability of cache blocks. The allocator returns a block ID to host-level instead of exposing the device's memory address to the host.

Third, OmniLib (discussed in in §3.2) uses a shim library to intercept POSIX I/O operations and convert them to OmniDev compliant NVMe commands. For HostCache management, we extend the scalable *jemalloc*[4] allocator for cache block allocation and release. When using a non-CXL near-storage device, the NVMe commands are copied to the per-file NVMe queues, which are later de-queued and dispatched by the OmniDev. In contrast, for *CXL.mem*, OmniLib directly accesses the device memory. We implement a CXL memory allocation semantics for OmniLib to register and allocate a CXL namespace that is shared across OmniLib and OmniDev.

Finally, **to emulate device memory and compute speeds, and CXL.mem latency and bandwidth,** we employ a two-step emulation. First, to emulate a slower device memory access from the host CPUs, we map device memory on a NUMA socket (node) remote to the host CPUs but local to the device CPUs. Precisely, in a system with two sockets, we allocate the host memory on NUMA *node*0, which is local to the host CPU and remote to the device CPUs. On the contrary, we allocate device memory on NUMA *node*1, which is local to the device CPU but remote to the host CPU. Next, to emulate device memory bandwidth, we throttle device memory's bandwidth using thermal throttling [6], lower device CPU speeds using frequency scaling, and add software latency to vary PCIe latency. In §6.4, we study the sensitivity of OmniCache on these hardware parameters.
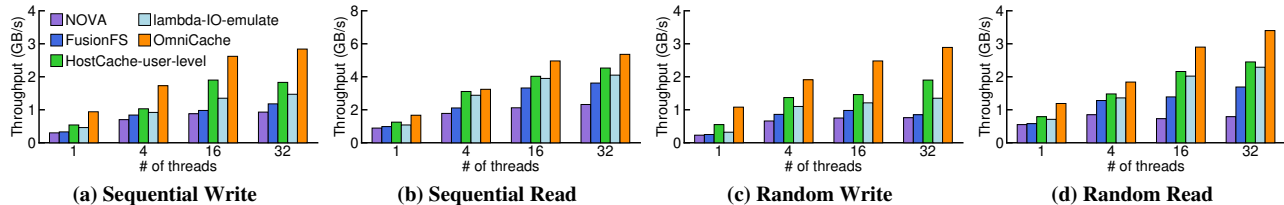
Figure 4: Microbenchmark. Threads use private files; workload size fixed at 64GB with 1KB I/O size and total cache size is 20GB.

6 Evaluation

We evaluate OmniCache to answer the following questions:

- How effective is OmniCache's collaborative use of Host-Cache and DevCache to improve I/O performance?
- Can OmniCache accelerate data processing with its concurrent and model-driven dynamic offloading?
- How does using OmniCache in conjunction with CXL impact the performance?
- What is the overall impact on real-world applications?

6.1 Experimental Setup

Environment: We use a dual-socket, 64-core, 2.7GHz Intel(R) Xeon(R) Gold platform with 32GB memory. For storage, we use a 512GB (4x128GB) Optane DC persistent memory with App-Direct (persistent) mode to represent the upcoming fast storage as well as 512GB NVMe SSD to study the benefit of OmniCache on slower storage. (see appendix appendix A.2.1). We emulate DevCache with 4GB of DRAM for caching and for OmniDev, we reserve 4 CPUs.

Methodology: For comparison, we consider the state-of-the-art PM OS file system, **NOVA**, and the near-storage file system, **FusionFS**, which lacks caching support (*FusionFS*). To understand the benefits and implications of host-only caching, we explore two configurations: (1) user-level host cache with OmniIndex atop FusionFS, referred to as *HostCache-user-level*; (2) emulated λ-I/O without FPGA but with OS caching (*lambda-IO-emulate*), which does not exploit near-storage cache (see Table 1). We emulate λ-I/O due to the unavailability of a customized hardware platform (Daisy/DaisyPlus OpenSSD) and OS (PetaLinux), and significant engineering challenges as highlighted by the authors [5]. Moving on to the OmniCache configuration, we begin by comparing our PCIe-based implementation without OmniDynamic, where offloading is solely determined by the data presence ratio (*OmniCache*). Subsequently, we compare *OmniCache-dynamic* to emphasize the impact of OmniDynamic on data processing performance. Finally, we evaluate *OmniCXL* to demonstrate the effect of CXL on performance. For all evaluation results, the total cache size is kept the same.

6.2 I/O Performance

We first evaluate I/O performance using sequential and random access I/O patterns. We vary workload threads from 1 to 32, each issuing 1KB I/O requests, resulting in a total workload size of 64GB. Figure 4 shows the cumulative throughput for private file access without file sharing, and Table 3 shows the benefits of using OmniIndex fine-grained concurrency when sharing files with multiple threads. Regarding

	FusionFS	HostCache-user-level	OmniCache
Readers	978	1893	2323
Writers	523	1223	1732

Table 3: File sharing. Results show aggregated throughput (MB/s) for 16 reader and 16 writer threads on a shared 64GB file.
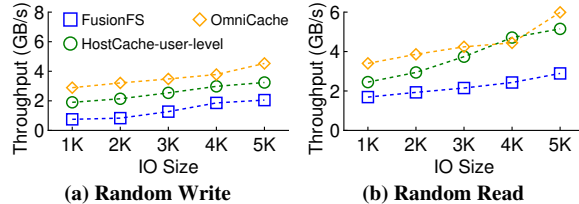

Figure 5: I/O Size Study.

caching, the *HostCache-user-level* and *lambda-IO-emulate* approaches employ a total of 20GB of host DRAM cache, while *OmniCache* configurations use 16GB HostCache and 4GB DevCache, a configuration used in prior systems [9, 33].

Observation: Figure 4a and 4c show results for sequential and random write workloads, respectively. As anticipated, NOVA and *FusionFS* lack caching and access storage for all I/O operations, which results in poor performance. Both *HostCache-user-level* with user-level caching and *lambda-IO-emulate* with OS caching show improvements but face high I/O stalls due to frequent cache eviction, particularly for random workloads. Similarly, for read workloads shown in Figure 4b and 4d, the host caching approaches, for each 1KB request, fetch 4KB blocks, increasing data movement cost, host-cache pollution, and suffer eviction stalls.

In contrast, OmniCache outperforms others by employing a collaborative approach that exploits host and device caches. Firstly, it adheres to the near-cache I/O principle, significantly reducing data movements between storage and host memory or between the host and device memory. For sequential access, DevCache identifies sequential access patterns, akin to Linux VFS, and preloads the entire 2MB data range (OmniIndex) to optimize data locality. However, it only returns the requested data (1KB) to the application, preventing I/O amplification and unnecessary data transfers. Subsequently, requests to the same 2MB range often result in cache hits. Secondly, OmniCache's collaborative caching performs writes or reads on both HostCache and DevCache, effectively minimizing write stalls and read times. As a result, OmniCache consistently outperforms *FusionFS* and *HostCache-user-level*, achieving performance gains of up to 2.53x and 1.52x, respectively.

I/O Size Sensitivity: We evaluate the impact of different I/O sizes by varying the request size from 1KB to 5KB while maintaining constant cache and workload sizes. This range encompasses both block-aligned and non-block-aligned
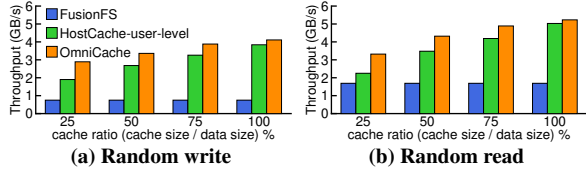
(a) Random write   **(b) Random read**

Figure 6: Cache Sensitivity Study. 32 microbenchmark threads with different cache ratio from 25% to 100% (maintaining equal total cache size for *HostCache-user-level* and *OmniCache*).
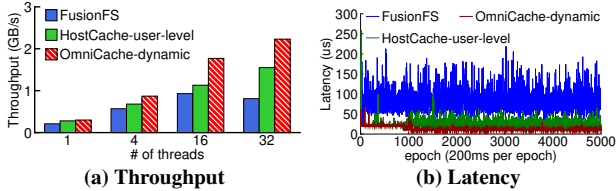

(a) Throughput   **(b) Latency**

Figure 7: I/O + Data Processing (Read-CRC-Write)

Figure 8: OmniDynamic Model Breakdown (Read-CRC-Write)

(a) Device CPU Frequency   **(b) Device Memory Bandwidth**
Figure 9: Model's Sensitivity for Read-CRC-Write

requests, mirroring real-world application behavior (Table 2). For instance, in RocksDB, numerous application I/O requests are not block-aligned. Figure 5 illustrates that *OmniCache* consistently outperforms other approaches for non-block-aligned requests across various workloads, reaping the benefits of near-cache I/O. For block-aligned (4KB) random write, *OmniCache* provides performance gains, attributable to its concurrent I/O and collaborative caching that reduces write stalls. For block-aligned random reads (e.g., 4KB), *OmniCache* performs similar to *HostCache-user-level*, as it moves the entire block to the host.

Impact of Data to Cache Size Ratios: We investigate the impact of the data-to-cache size ratio on the throughput of random read and write workloads. Figure 6 displays this impact, with the x-axis ranging from 25% to 100% cache ratio. A 25% ratio implies that the data size is four times larger than the combined HostCache and DevCache sizes. At lower ratios, *HostCache-user-level* experiences frequent evictions and thread stalls. In contrast, OmniCache smoothly transitions application threads to utilize DevCache when HostCache reaches its capacity. It then initiates background eviction of HostCache before switching back to HostCache. Even at a 100% cache ratio, OmniCache outperforms others by minimizing data movement using near-cache I/O principle.

6.3 Data Processing with OmniCache

We evaluate the effectiveness of collaborative processing and dynamic offloading with OmniCache for I/O and compute-intensive read-CRC-write workload. Thread randomly reads 4KB data blocks, calculates checksum, and writes it back.

Throughput Analysis: Figure 7a illustrates that *FusionFS* encounters NVMe command communication and data copy overheads. It necessitates offloading each request to the device, performing computations on the device, and subsequently writing the data back to device storage. On the other hand, *HostCache-user-level* operates more efficiently when requests are served from the host cache, enabling direct execution on the host. However, in cases of cache misses, *HostCache-user-level* incurs data movement overhead between storage and host memory, which hinders computation.
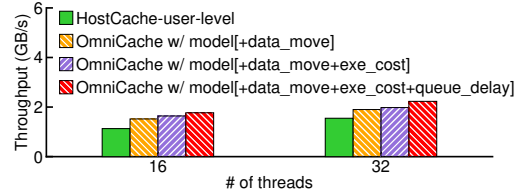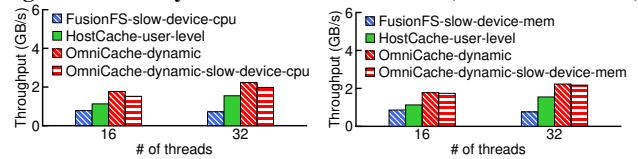
In contrast, *OmniCache-dynamic* dynamically offloads data processing operations by efficiently considering multiple factors. Figure 7a illustrates that *OmniCache-dynamic* significantly enhances performance, especially under heavy workload scenarios. This improved performance results from its model-driven approach, which continuously monitors execution time, queue latency, and other factors to dynamically determine the optimal offloading location.

Latency Analysis: In Figure 7b, we examine latency variations in the read-CRC workload. HostCache handles write requests well initially but experiences fluctuations and delays due to evictions after epoch 1000, resulting in longer queue delays. Additionally, execution costs vary due to higher processor cache misses and I/O frequency. In contrast, *OmniCache-dynamic*'s model-driven approach, which considers factors like data distribution ratio, queue length, and execution costs, maintains lower and stable latency, outperforming *HostCache-user-level* by up to $1.42\times$.

6.4 OmniDynamic Model Effectiveness

We empirically validate the effectiveness of OmniDynamic model by deciphering the impact of model parameters and the sensitivity to hardware speeds.

Model Performance Breakdown: Figure 8 shows performance analysis by gradually incorporating different parameters of the model (data movement, execution time, queuing delays) and understanding their impact on the read-CRC-write workload. First, the decision to offload to device or process in host, is significantly influenced by data fraction (R_h or R_d) in HostCache or DevCache (*model[+data_move]*). Second, the execution time (Ed_{avg} or Eh_{avg}) fluctuates, impacted by factors like data presence in the host and device processor caches that can accelerate execution (*model[+data_move+exe_cost]*). Finally, overheads of queuing delay ($Cmd_{avg} * Q_{len}$) becomes particularly pronounced for higher application thread count and frequent background eviction (*model[+data_move+exe_cost+queue_delay]*).

Sensitivity to Hardware Parameters: To understand the *OmniCache-dynamic* model's sensitivity toward device CPU speed and low-bandwidth memory, we show the performance of OmniCache with slower CPU (Figure 9a). Due

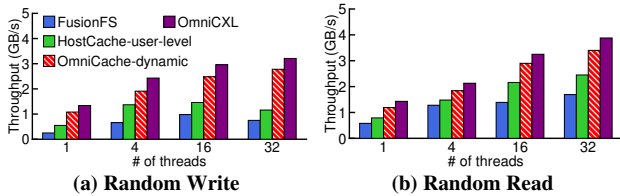**(a) Random Write**   **(b) Random Read**

Figure 10: OmniCache with CXL

Vary Device Hardware Parameters			
Device CPU Frequency	2.7 GHz	2.0 GHz	1.2 GHz
Device Memory B/W	120 GB/s	60 GB/s	16 GB/s
PCIe Latency	900ns	1200ns	1500ns
# of ops. executed in host	**12.52M**	**14.67M**	**15.24M**
# of ops. executed in device	**4.19M**	**1.90M**	**834.32K**

Table 4: OmniDynamic Model's Sensitivity Analysis

to space constraints, we only consider *FusionFS* and Omni-Cache. The device CPU is throttled to 1.2GHz for FusionFS (*FusionFS-slow-device-cpu*) and OmniCache (*OmniCache-dynamic-slow-device-cpu*) compared to 2.7GHz for host CPUs. Similarly, in Figure 9b, the device memory is throttled to 16GB/s for FusionFS (*FusionFS-slow-device-mem*) and OmniCache (*OmniCache-dynamic-slow-device-mem*) compared to 120GB/s for host DRAM ($8\times$ bandwidth reduction). Despite slower device CPUs and reduced memory bandwidth, OmniCache provides up to 1.22X gains over HostCache by dynamically distributing work across the host and the device.

To comprehend the model's sensitivity and work distribution, in Table 4, we vary the device CPU, memory, and PCIe latency values and monitor *OmniCache-dynamic*'s offloading decision. As we gradually reduce the device CPU frequency, lower the memory bandwidth, and increase PCIe latency, *OmniCache-dynamic* increases operations on the host rather than indiscriminately offloading them to the device.

6.5 CXL.mem enabled OmniCache

Figure 10 shows the benefits of using CXL.mem for random access workloads. While OmniCache without CXL incurs overheads from data copies between the host and the device and queuing delays, *OmniCXL* directly accesses DevCache with CPU loads/stores after acquiring ownership of a range. This improves performance by diminishing these overheads. Furthermore, *OmniCXL* reduces CPU polling cost for request completion, all leading to 2.76X and 1.21X gains over *HostCache-user-level* and *OmniCache*, respectively.

6.6 Real-World Applications

We next evaluate the benefits of OmniCache on real applications. **LevelDB:** LevelDB is a widely-studied LSM-based persistent key-value store [3]. We modify LevelDB's *append→checksum→write* sequence by introducing the append-CRC-write operation and *read→checksum* sequence with read-CRC, similar to *FusionFS* (11 LOC changes). The checksum operations are used in LevelDB to avoid frequent commits on SST files [9]. We run experiments using the widely-used YCSB cloud workload [14] that comprises six distinct access patterns (A-F), each with differing read/write ratios and exhibits a Zipfian distribution with access locality.
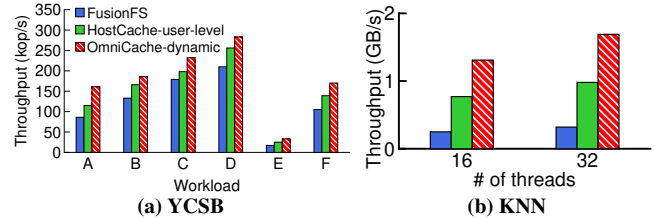

(a) YCSB   **(b) KNN**

Figure 11: Real-World Applications.

We use 512B value sizes, 40 million keys, and 32 threads.

Performance: As shown in Figure 11a, OmniCache outperforms across all workloads. Particularly, write-intensive A and F workloads show maximum gains over *HostCache-user-level* attributed to (1) near-cache I/O that reduces data movement for non-block aligned requests, (2) collaborative caching that minimizes CPU stall time, and (3) dynamic offloading to effectively use of host and device resources. Over FusionFS, OmniCache shows up to 1.92X gains. db_bench [8] for random workloads show even higher gains (see appendix A.2.2).

Nearest Neighbor Search (KNN): Next, we evaluate OmniCache using a complex KNN workload, utilizing an implementation from prior work [33]. However, we deviate from their assumption of the entire workload fitting into device memory. Instead, we employ a 128GB workload, with 20GB *HostCache-user-level* or 16GB HostCache and 4GB DevCache for OmniCache. To handle datasets larger than the cache size, the application divides the file into shards and, for each shard, performs distance calculations (see §4.3), merging the per-shard distances for KNN prediction.

As Figure 11b shows, *FusionFS*, without caching, performs poorly as KNN execution requires reading each shard from storage to the device. *HostCache-user-level* offers marginal improvement but encounters significant data movement and eviction overheads as the data size surpasses the cache. In contrast, OmniCache effectively and concurrently utilizes both host and device for *read-cal_distance-predict*, resulting in performance gains of up to 5.15X over FusionFS.

7 Conclusion

We develop OmniCache, a collaborative caching design to leverage host and device memory as cache to accelerate I/O and data processing. OmniCache achieves this through scalable indexing, concurrent caching and processing support, and a dynamic model-centric offloading technique leading to substantial performance gains on both microbenchmarks and applications.

Acknowledgements

References

[1] CXL Specification. https://www.computeexpresslink.org/download-the-specification.

[2] Gluster. https://www.gluster.org/.

[3] Google LevelDB . http://tinyurl.com/osqd7c8.

[4] jemalloc. https://jemalloc.net/.

[5] λ-IO GitHub. https://github.com/thustorage/lambda-io#building-and-running.

[6] memhog - Allocates memory with policy for testing. https://man7.org/linux/man-pages/man8/memhog.8.html.

[7] rbtree based interval tree as a prio_tree replacement. https://lwn.net/Articles/509994/.

[8] RocksDB db_bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools/.

[9] FusionFS: Fusing I/O operations using CISCOps in firmware file systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 297–312, Santa Clara, CA, February 2022. USENIX Association.

[10] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, Portland, OR, 2011.

[11] ARM. https://www.arm.com/solutions/storage/computational-storage.

[12] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

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

[14] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[15] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications. *ACM Trans. Storage*, 16(4), October 2020.

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

[17] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 173–187. USENIX Association, July 2020.

[18] Suhas Jayaram Subramanya, Fnu Devvrit, Harsha Vardhan Simhadri, Ravishankar Krishnawamy, and Rohan Kadekodi. Diskann: Fast accurate billion-point nearest neighbor search on a single node. *Advances in Neural Information Processing Systems*, 32, 2019.

[19] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 45–51, 2022.

[20] Myoungsoo Jung. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '22, page 45–51, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Sudarsun Kannan, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Yuangang Wang, Jun Xu, and Gopinath Palani. Designing a True Direct-access File System with DevFS. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, FAST'18, pages 241–255, Berkeley, CA, USA, 2018. USENIX Association.

[22] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 993–1005, Boston, MA, July 2018. USENIX Association.

[23] Hyojun Kim, Sangeetha Seshadri, Clement L Dickey, and Lawrence Chiu. Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 33–45, 2014.

[24] Jinhyung Koo, Junsu Im, Jooyoung Song, Juhyung Park, Eunji Lee, Bryan S. Kim, and Sungjin Lee. Modernizing file system through in-storage indexing. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 75–92. USENIX Association, July 2021.

[25] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

[26] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Jing Liu, Anthony Rebello, Yifan Dai, Chenhao Ye, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Scale and performance in a filesystem semi-microkernel. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 819–835, New York, NY, USA, 2021. Association for Computing Machinery.

[28] NVIDIA Mellanox BlueField DPU. https://www.mellanox.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[29] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, Broomfield, CO, 2014.

[30] Nicholas Rauth. A network filesystem client to connect to SSH servers. https://github.com/libfuse/sshfs.

[31] Yujie Ren, Changwoo Min, and Sudarsun Kannan. {CrossFS}: A cross-layered {Direct-Access} file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154, 2020.

[32] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, page 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

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

[34] Samsung. Samsung Key Value SSD. https://www.samsung.com/semiconductor/global.semi.staticSamsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[35] ScaleFlux. https://scaleflux.com/.

[36] Seagate RISC-V storage solution. https://www.seagate.com/innovation/risc-v/.

[37] Yasas Seneviratne, Korakit Seemakhupt, Sihang Liu, and Samira Khan. Nearpm: A near-data processing system for storage-class applications. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 751–767, New York, NY, USA, 2023. Association for Computing Machinery.

[38] SNIA. SNIA Computational Storage Technical Work Group (TWG).

[39] Mohammadreza Soltaniyeh, Veronica Lagrange Moutinho Dos Reis, Matt Bryson, Xuebin Yao, Richard P. Martin, and Santosh Nagarakatte. Near-storage processing for solid state drive based recommendation inference with smartssds®. In *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, ICPE '22, page 177–186, New York, NY, USA, 2022. Association for Computing Machinery.

[40] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014.

[41] Matthew Wilcox and Ross Zwisler. Linux DAX. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[42] Mark Wilkening, Udit Gupta, Samuel Hsia, Caroline Trippel, Carole-Jean Wu, David Brooks, and Gu-Yeon Wei. Recssd: near data processing for solid state drive based recommendation inference. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 717–729, 2021.

[43] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, FAST'16, Santa Clara, CA, 2016.

[44] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. $\lambda - io$: A unified IO stack for computational storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 347–362, Santa Clara, CA, February 2023. USENIX Association.

[45] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, pages 2195–2207, New York, NY, USA, 2021. Association for Computing Machinery.

	FusionFS w/ NVMe Storage	HostCache-user-level	OmniCache
Random Read	731	2153	2521
Random Write	990	1573	1982

Table 5: Microbenchmark on NVMe Storage. Results show aggregated throughput (MB/s) for 32 benchmark threads.

A Appendices

A.1 Discussion

We have implemented various functions, including checksum, compression, nearest-neighbor search, and text search operations (not all shown due to space limitations). However, it is important to note that, like most existing systems, OmniCache assumes that OmniDev already incorporate these functions. Even previous near-storage systems [9, 10, 33], and those that use eBPF-based offloading [44] require device-level modifications and frequent kernel traps, which are not ideal for I/O-intensive applications. Developing a more generic offloading mechanism without requiring application changes requires compiler support, and is a complex task that falls outside the scope of this paper. In addition, our future work will explore using CXL.mem to enable memory-mapping (mmap()) support. To the best of our knowledge, existing systems also lack this particular feature.

A.2 Additional Performance Evaluation

A.2.1 Sensitivity to Slower NVMe Storage

To understand the impact of OmniCache when using slower storage media, NVMe-based SSD, we use the same experimental configuration and microbenchmarks to study the throughput. As shown in Table 5, OmniCache shows an even higher performance gain of 3.24x over *FusionFS* and 1.21x over *HostCache-user-level*. Notably, the gains are high compared to PM-based storage. These gains highlight the importance of collaborative cache use for slower storage. We observe performance gains even for data processing workloads (not shown due to space constraints).

A.2.2 Impact of OmniCache for LevelDB's db_bench

In order to assess the efficiency of the collaborative caching design offered by OmniCache, we also evaluate the random write and random read workload in Figure 12 using the widely-used db_bench for 1 million key-value pairs and 32 application threads with 4KB value size. OmniCache delivers higher performance across all workloads. These enhancements can be attributed to collaborative caching for I/O operations. In addition, OmniCache's dynamic offloading further amplifies these gains by ensuring optimal resource utilization between the host and device.
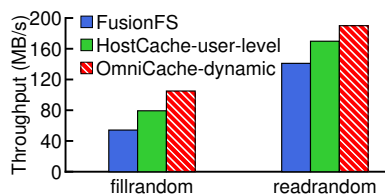

Figure 12: LevelDB (db_bench)

B Artifact Appendix

Abstract

The OmniCache artifact is the practical implementation of the collaborative caching design presented in this paper, aimed at optimizing I/O and data processing by utilizing both host and device memory as caches.

Scope

The artifact enables validation of the benefits of collaborative caching for concurrent I/O, the impact of dynamic model-driven offloading on data processing, and showcases the extensibility of OmniCache with CXL. OmniCache is licensed under Apache License 2.0.

Contents

The OmniCache artifact comprises user-level library (OmniLib) and the OS component for emulating near-storage device (OmniDev), which are required for the execution. The artifact comprises real-world applications (LevelDB, YCSB workload and KNN) besides microbenchmarks, as shown in the paper. The artifact includes steps to compile the user-level library, the OS, microbenchmarks, and applications, and steps to run these workloads.

Hosting

The artifact is available on GitHub: omnicache-fast24-artifacts.

Requirements

Our artifact is based on Linux kernel 4.15.18 and it should run on any Linux distribution. The current scripts are developed for Ubuntu 18.04.5 LTS. Porting to other Linux distributions would require some script modifications. Our artifact requires a machine equipped with Intel Optane memory.

Evaluation

We provide a comprehensive step-by-step README on GitHub to reproduce the experiment in the paper. As a brief overview of the evaluation, we illustrate how to execute a simple microbenchmark with OmniCache. More evaluations can be found on our GitHub page.

Before running, we assume the modified kernel (OmniDev) is installed, NearStorageFS is mounted on the machine (please see the README file) and the current work directory is in the project's root directory.

1. First, compile and install the user-level library (OmniLib):
```
$ source ./scripts/setvars.sh;
$ cd $LIBFS;
$ source scripts/setvars.sh
$ make && make install
```
2. Next, to run a simple micro benchmark:
```
$ cd $BASE/libfs/benchmark/;
$ mkdir build && make
$ ./scripts/run_omnicache_quick.sh
```
Expect output will be similar to "Benchmark takes 0.97 s, average throughput 4.45 GB/s". If the output matches the above, your environmental settings are appropriately configured.