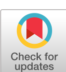

Exploring the Asynchrony of Slow Memory Filesystem with EasyIO

Bohong Zhu
zhubohong12@stu.xmu.edu.cn
Xiamen University

Youmin Chen*
chenyoumin@tsinghua.edu.cn
Tsinghua University

Jiwu Shu
shujw@tsinghua.edu.cn
Xiamen University, Minjiang University & Tsinghua University

Abstract

We introduce EasyIO, a new approach to explore *asynchronous I/O* on filesystems designed for (disaggregated) non-volatile memories to improve CPU efficiency. EasyIO offloads expensive memory movement operations to the on-chip DMA engine and harvests the unleashed CPU cycles by transparently interleaving asynchronous I/Os with fine-grained application tasks. We further adopt a completion buffer-centric design to improve EasyIO's efficiency and schedulability; internally, *orderless file operation* and *two-level locking* are incorporated to break the serial order between file metadata and data, thus accelerating read and write operations and defusing deadlock risks. EasyIO also introduces a traffic-aware *channel manager* to fulfill the diverse performance goals of applications. Extensive experimental results show that, compared to conventional synchronous filesystems, EasyIO significantly reduces CPU consumption (using less than 88% of cores at most) while achieving comparable peak bandwidth; EasyIO also achieves 1.03-2.3× speedups across eight real-world applications. When achieving these goals, EasyIO exhibits higher but tolerable latencies for read operations due to the task interleaving.

1 Introduction

With the proliferated demand for real-time data processing (e.g., data analysis [18, 55], web service [31], and graph processing [24, 38, 56]), traditional DDR-based DRAM falls short due to its limited capacity scalability and high monetary cost. Regarding this, the data center infrastructure is exploring new system architectures and memory technologies. Among them, non-volatile memory technologies, such as those provided by Intel's Optane `DCPMM` [13] and Samsung's memory-semantic SSDs [17], are an appealing building block for efficient data storage due to its high density and low cost; memory disaggregation decouples the compute and memory into resource pools and connects them via cache-coherent links (e.g., CXL [4]) to enjoy higher resource utilization and elasticity (independent scaling). Compared to traditional memories, (disaggregated) non-volatile memories are typically larger, cheaper, and slower, and thus form a new layer in the memory hierarchy, which we call *slow memory*.

A line of recent research work [19, 27, 28, 30, 43, 44, 54, 68, 74, 78] proposes to abstract slow memory with *files* due to its familiar programming manner, mature software ecosystem, and full-fledged permission control. However, these filesystems provide only the synchronous I/O interface since memory `load/store` instructions are inherently synchronous. Besides, a `load` request that misses in the on-chip cache and goes to the slow memory often stalls the processor instruction pipeline for up to hundreds of *ns*, incurring heavy CPU consumption [23]. As we will show in § 2.1, a slow memory-based filesystem can devote up to 95% of CPU cycles to memory movement operations. This significantly limits the efficiency of modern out-of-order (OOO) processors and thus exacerbates the storage tax problem [50].

This paper presents a fundamentally different approach to address this problem by exploring the *asynchrony* of filesystems for slow memory. Actually, asynchronous I/O has been extensively used for managing external storage and network devices (e.g., `libaio` [16], `io_uring` [2], SPDK [3], and DPDK [6]) due to its high CPU utilization – the processor core issues an I/O command to the device and waits for I/O completion asynchronously, during which the CPU cycles can instead be used to run applications. In particular, Intel introduced I/O Acceleration Technology (I/OAT) [12], and more recently, the Data Streaming Accelerator (DSA) [11], which can offload memory operations from the CPU to an on-chip DMA engine. The DMA engine[1] is programmed and operated in the same way as the PCIe-attached devices, enabling us to access slow memory in an asynchronous manner.

However, memory-level asynchrony does not directly indicate high CPU utilization. Filesystems designed for byte-addressable slow memory typically face I/Os that are much finer in size (ranging from a few to tens of KBs), where an asynchronous I/O can be finished within μs-scale time

*Youmin Chen is the corresponding author.

[1] *On-chip DMA engine* and *DMA engine* are used interchangeably in the paper to refer to the offloading engine provided by I/OAT or DSA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EuroSys '24, April 22–25, 2024, Athens, Greece

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0437-6/24/04...$15.00
https://doi.org/10.1145/3627703.3629586

windows. As a result, offloading memory operations to the on-chip DMA engine makes CPU execution highly fragmented. Such short time windows are hard to be harvested by the operating system since rescheduling a task incurs millisecond-scale scheduling latencies [31, 60]. An alternative is exposing such asynchrony to applications and letting the application interleave asynchronous I/Os with fine-grained computation tasks, which, however, significantly increases the complexity of application codes [47] and raises hard scheduling problems – executing too long tasks makes I/O completion checking to be delayed, diminishing the low-latency advantage, while executing too short tasks leaves CPU cycles underutilized.

We introduce *schedulable asynchronous I/O*, or EasyIO, to address the above challenge. Motivated by the recent trend of using light-weight userspace threads (uthreads) to run application logic [9, 22, 36, 60, 62, 65, 66, 77], we make the fragmented execution windows easily harvestable through the careful coordination between asynchronous I/Os and uthread scheduling. Specifically, uthreads are multiplexed on physical cores and scheduled by userspace runtime (i.e., the scheduler). Whenever a uthread issues an asynchronous I/O, the CPU core context switches to run the next runnable uthread, and switches back when the I/O finishes. In this way, the complexity of interleaving I/O and computation is hidden from the application to the runtime and the CPU efficiency is increased dramatically. While the idea of combining asynchronous I/Os and uthread scheduling is intuitive, applying EasyIO into a filesystem raises efficiency and liveness issues, and we further incorporate several techniques to address them.

First, for durability and atomicity reasons, EasyIO requires two interactions with the filesystem to complete an asynchronous I/O, i.e., one for DMA-offloaded data copy and the other for metadata commit. This inevitably incurs high latencies. Even more problematic, intermediate scheduling between the two interactions may let a CPU core acquire a file lock twice, leading to deadlock risks. We adopt a novel use of the completion buffer associated with the DMA engine to allow data and metadata to be performed simultaneously. Originally, the DMA engine modifies the completion buffer to describe the completion of a memory copy; we extend its semantics to describe the completion of `write()` operations as well. By placing the completion buffer in a predefined persistent region with its stored value increased monotonically, EasyIO breaks the serial order between updating file metadata and data, enabling them to be processed in parallel (§ 4.2). The completion semantics also guides the design of a *two-level lock* to avoid deadlocks. Before the DMA-offloaded data copy completes, EasyIO can release the file lock in advance along with the metadata commit; to acquire the lock, one first grabs the file lock and further checks the completion buffer to determine whether a file is safely locked (§ 4.3).

Second, offloading memory operations exposes the hardware complexity (e.g., channels, command queues, and registers as in I/OAT) to the software, which can easily lead to inefficient use of the DMA engine. Through an in-depth analysis of the hardware behavior when offloading memory operations to DMA engines, we derive a number of important design guidelines – among which some are even counterintuitive. For instance, using multiple DMA channels does not necessarily lead to higher bandwidth, which instead causes performance degradation; using fewer DMA channels, however, incurs serious interference between concurrent requests due to head-of-line blocking in the command queue and bandwidth contention. We introduce a *channel manager* that carefully mediates the interaction between concurrent DMA requests and channels to fulfill the diverse performance goals of latency-critical (`L-apps`) and bandwidth-oriented (`B-apps`) applications (§ 4.4). Whenever an `L-app` fails to meet its latency SLOs, the channel manager throttles `B-apps` by manipulating the related DMA channels at μs-scale time intervals.

EasyIO does not depend on a specific filesystem; in our implementation, we apply EasyIO to NOVA [74], a time-tested kernel persistent memory filesystem with standard POSIX APIs and strict data persistence[2]. We replace its data movement operations in `read()` and `write()` and make minor modifications to the file metadata format and recovery logic (less than 50 lines of code). To demonstrate the efficiency of EasyIO, we conduct extensive tests with both microbenchmarks (FxMark [58]) and real-world applications. Experimental results show that, compared to conventional synchronous filesystems, EasyIO significantly reduces CPU consumption. Specifically, it uses up to 88% fewer CPU cores to achieve a similar peak write throughput than NOVA. EasyIO delivers lower average and 99th percentile latencies than NOVA for write operations, but incurs higher read latencies due to the interleaving of computation and I/Os. EasyIO also achieves 1.03-2.3× speedups across eight real-world applications. EasyIO is novel in the following ways:

- As far as we know, EasyIO makes the first attempt to explore the asynchrony of slow memories by combining uthread scheduling and asynchronous I/O, which makes μs-scale execution windows easily harvestable.
- We further improve EasyIO's efficiency and schedulability with a completion buffer-centric design by introducing orderless file operation and two-level locking.
- We introduce a channel manager to enable EasyIO to fulfill different applications' diverse performance goals.

2 Motivation and Background

In this section, we briefly introduce the basic concepts of slow memory storage systems and userspace threads, and then empirically analyze the overhead of data movement and the performance of I/OAT.

[2]Both data and metadata operations ensure durability and atomicity.

2.1 Slow Memory Storage Systems

Slow memory, such as non-volatile memory (NVM) that is disaggregated or not, is connected to the processor directly by either the memory bus or cache-coherent interconnects like Compute Express Link (CXL) [4]. Slow memory is expected to be with large capacity (up to TBs), data durability, low monetary cost, and byte-addressability accessed via `load/store` instructions of CPU. In this paper, we focus on Intel's Optane `DCPMM`, the first (also the only) commercially available non-volatile memory product released in 2019, but the techniques proposed in our paper can be equally applied to other byte-addressable NVMs such as ReRAM [20], memory-semantic SSDs [17]), and others.

Compared to DRAM memories, Optane `DCPMMs` have 2-3.7× higher latency and 1/3rd bandwidth for loads, while stores have the same latency but 1/6th bandwidth [42, 44]. With these performance features in mind, many research work focuses on building scalable and performant filesystems on top of it [27, 28, 30, 35, 43, 44, 51, 74]. These systems retain the standard file interfaces but greatly improve the software efficiency by removing the page cache to enable direct access (DAX), adopting lightweight journaling mechanisms [35] and using fine-grained locks [27].

However, restricted by the blocking nature of `load/store` instructions, all existing NVM filesystems only provide synchronous interfaces to applications, incurring high CPU overhead. We use the FxMark benchmark [58] to generate single-threaded read and write requests with I/O sizes ranging from 4KB to 64KB, and investigate the latency breakdown of NOVA [74] when processing these requests (detailed experimental setup will be shown in § 6.1). As shown in Figure 1, metadata, memcpy, indexing, and syscall & VFS correspond to the latencies spent on metadata update, file data movement, file indexing, and OS-level overhead for system calls and virtual file system, respectively. We find that up to 95% and 63% of CPU cycles are spent on data copy operations for `read()` and `write()`, respectively, as the I/O size grows to 64KB. Moreover, it is even more harmful when cross-socket data movement on Optane is involved as many recent work revealed [27, 76]. Similar problem does not happen on SSDs or HDDs since the device is responsible for data movement via on-device DMA engine, during which the CPU core can be scheduled to do other useful work.

2.2 Empirical Study of On-Chip DMA Engine

Many modern processors have included on-chip DMA engines to offload expensive memory movement operations. Intel introduced I/O Acceleration Technology (I/OAT) to Xeon 5100 series processor-based platforms as early as in 2006 [12]. I/OAT is actually a set of technologies; among them, the QuickData Technology enables data copy by an on-chip DMA engine instead of the CPU. In response to the growing need for hardware offloading, Intel recently introduces the Data Streaming Accelerator (DSA) in its 4th generation Xeon Scalable CPUs (codename Sapphire Rapids). Studies report that DSA has better efficiency and lower energy consumption [11]. Similarly, AMD's 2nd generation EPYC CPUs are also equipped with such DMA engines [64].

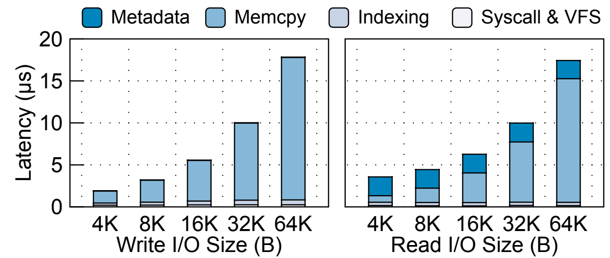

Figure 1. Latency breakdown of NOVA [74].

Workflow. The basic workflow of handling a memory copy request via the DMA engine goes with the following steps (takes I/OAT as an example). The CPU core first prepares a DMA descriptor containing the metadata required for the DMA engine, including the source and the destination memory addresses and the size that the core intends to copy. Then, the CPU core inserts the DMA descriptor into a hardware queue (a contiguous memory buffer), and submits it via memory-mapped I/O (MMIO) registers. Once the DMA engine finishes the DMA request, it claims the completion by updating the completion buffer (a 64-bit integer) to point to the latest finished DMA descriptor in the hardware queue.

Both I/OAT and DSA contain multiple channels (or processing engines in DSA) and support batch submission of DMA requests. However, I/OAT requires that the source and destination memory buffer to be physically contiguous and pinned before the DMA engine uses them. Instead, DSA supports shared virtual memory (SVM) that performs address translation with the on-device address translation cache (ATC); therefore, DSA allows applications to submit DMA requests directly with virtual addresses and memory pinning is not required, substantially lowering the offloading overhead.

We use the combination of I/OAT and Optane `DCPMM`, a representative (also the only available) hardware setting for us, to study asynchronous I/Os on slow memory. Su et al. [69] has already conducted a performance review and derived some interesting findings (e.g., uneven advantages between reads and writes over the CPU). However, they mainly focused on the latency under low concurrency, and we extend the experiment to explore its performance limits across multiple channels and cores. Accordingly, we implement a test tool that can issue read (write) requests from (to) Optane `DCPMMs` through the DMA engine or CPU-involved `memcpy` by tuning the number of CPU cores, I/O sizes, batch size, and DMA channels. The experiment is conducted on one NUMA node with 3 Optane `DCPMMs` attached.

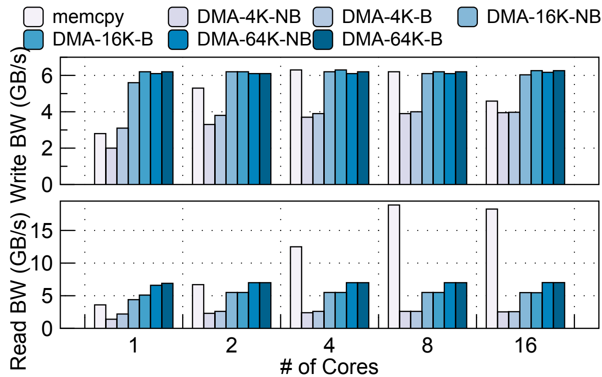

Figure 2. Bandwidth comparison of `memcpy` and on-chip DMA engine. *`B` and `NB` indicate 'batch-size = 4' and 'no batch' respectively. One channel used for DMA; I/O sizes larger than 64KB are not shown since they do not increase bandwidth; for the same reason, we only show the `memcpy` bandwidth with the 4KB I/O size.*

Figure 2 compares the performance that the DMA engine and `memcpy` can achieve when copying data between the DRAM and the DCPMM as we increase the number of cores, I/O size, and batch size. Here we only use one DMA channel for I/OAT (multi-channels will be tested later) and draw the following basic conclusions. ❶ On-chip DMA engine achieves higher CPU efficiency, especially for writes. DMA can saturate Optane `DCPMM`'s bandwidth for writes with only one core, while `memcpy` requires multiple cores. ❷ On-chip DMA engine has asymmetric advantages over CPU for reads and writes. Specifically, DMA fails to reach the peak bandwidth for reads (63% lower than `memcpy`), and it is more CPU-efficient than `memcpy` only when using larger I/O sizes and fewer cores. ❸ DMA is less efficient with small I/Os. When using 4KB I/Os, DMA always underperforms `memcpy` even when batching is enabled. ❹ `memcpy`'s performance declines for writes as the number of cores increases (reported by many past work [27, 76]) while the DMA engine does not have similar problems. Several extra takeaways are highlighted below, which provide us with important design guidelines.

Multi-channel is not always beneficial. We further evaluate the performance of the DMA engine by increasing the number of channels and the results are shown in Figure 3. Here we use 16 cores to submit I/O requests concurrently to ensure that DMA channels are saturated. We observe that multi-channel has a differentiated impact on reads and writes. For writes, using 4 channels achieves an optimal bandwidth with a 4KB I/O size, adding more channels, instead, can actually lead to slight performance degradation; for larger I/O sizes, the bandwidth decreases monotonically as the number of channels increases. The performance of read operations never declines and peaks at 2 - 4 channels with larger I/O sizes.

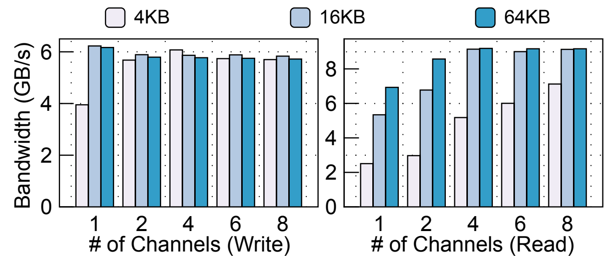

Figure 3. Bandwidth with a varying number of DMA channels. *16 cores are used to issue DMA requests concurrently.*

On-chip DMA incurs serious latency spikes. Here we let a foreground program issue 64KB reads with the DMA engine and measure its latency as a background program conducts bulk data movement (2MB) periodically via `memcpy` or DMA (to simulate garbage collection). When using the DMA engine, the foreground and background programs can either share DMA channels or use separate ones (denoted as `DMA-SH` and `DMA-EX`, respectively). As we can see from Figure 4, foreground programs exhibit more than a 2× increase in latency when background instances switch from `memcpy` to DMA, and the latency jitters much worse when they share DMA channels. The main reason is two-fold: first, the DMA engine consumes device bandwidth disproportionately, leading to starvation of some DMA channels; second, `DMA-SH` further serializes concurrent requests in the same hardware queue, incurring serious head-of-line blocking.

In conclusion, with the unique hardware features of the DMA engine in mind, we should devise appropriate strategies for using it more efficiently.

2.3 Lightweight Userspace Threads

Operating system is originally designed for slow disks and Ethernet, and threads (or kernel threads) are typically preemptively multitasked on physical cores and scheduled by a kernel scheduler at coarse-grained *millisecond* timescales. Advances in networking and storage technologies increase the need for scheduling cores at μs timescales, and lightweight userspace threads (uthreads) become increasingly popular. Uthreads are essentially similar to kernel threads – both of them own separate running states including the stack, CPU registers, and thread-local global variables. However, uthreads are implemented entirely in userspace, where context switching does not require any interaction with the kernel and is thus extremely efficient. The uthread itself can perform context switches with an embedded userspace runtime, which locally saves the CPU registers of the currently executing uthread and then loads the registers of the next runnable one. Normally, uthread is uninterruptible unless it invokes a `yield` function actively when blocking on I/Os or locks, which makes CPU execution be transferred to the next uthread. Preemptive uthread scheduling still requires kernel involvement to

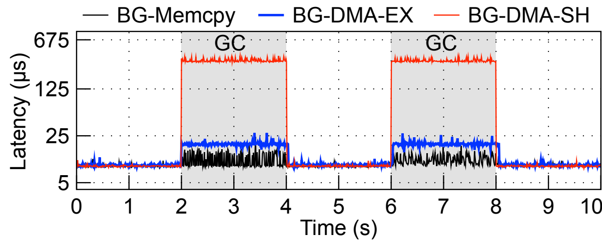

Figure 4. Interference between foreground and background programs. *Log-scale; `EX` and `SH` indicate whether foreground and background programs use separate or shared DMA channels. `GC` indicates the Garbage Collection.*

send privileged inter-core signals. Many recent core scheduling systems are based on uthreads to achieve high CPU efficiency [36, 57, 60–62].

3 Challenges

Our overarching goal is to optimize the CPU efficiency of filesystems by exploiting the *asynchrony* of slow memory. Specifically, we introduce EasyIO that offloads expensive memory movement operations to the on-chip DMA engine and harvests the released CPU cycles that will be instead wasted for busy-polling completions.

In EasyIO, application tasks run inside uthreads, which are scheduled with a userspace scheduling runtime. Whenever a uthread issues an asynchronous I/O request to the filesystem, it returns back to the runtime immediately to let the CPU core run other colocated uthreads. EasyIO allows CPU cores to always do useful work, and thus increase CPU efficiency. However, combining asynchronous I/O and uthread scheduling in EasyIO raises efficiency and liveness issues.

Multiple interactions with the filesystem. To process a `write()` operation, the filesystem needs to update data pages and the related block mappings (which map file pages to their physical addresses). For example, in NOVA [74] which provides strict data persistence [44], data pages are written in a copy-on-write (CoW) manner, then the block mappings are updated atomically by appending a log entry to commit the updates made on the data pages. As we can see, metadata updates and data page updates are strictly ordered into separate stages; with this, the filesystem determines whether a `write()` is finished relying on the completion of metadata updates. However, EasyIO allows asynchronous write operations to be returned in advance before the I/O completes, at which the metadata has not been modified yet. This requires the runtime to interact with the filesystem again to modify file metadata. An alternative way is using a background thread to update metadata when the I/O finishes, which, however, is contradictory to our design goal of saving CPU cycles.

Uthread scheduling leads to deadlocks. Normally, a file should be kept locked during the whole operation to prevent others from accessing inconsistent data. In asynchronous I/O mode, unlocking a file is conducted when committing metadata, which appears in a different stage than locking. Scheduling between the two stages may let a CPU core access the same file again for another uthread, while the file is still locked by a previous uthread, leading to deadlocks.

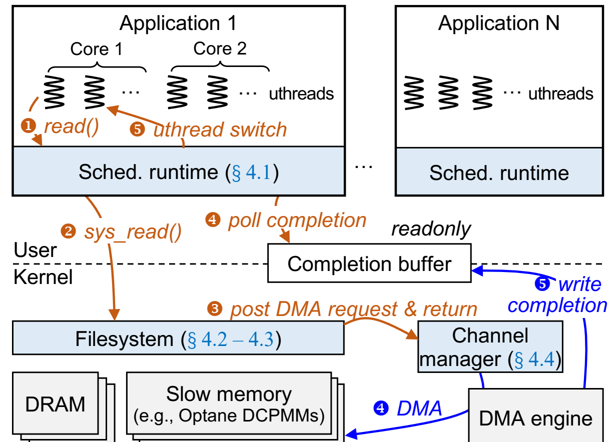

Figure 5. Overall architecture of EasyIO. *Highlighted steps show how EasyIO handles a `read()` syscall asynchronously.*

DMA bandwidth regulation is hard. As shown in § 2.2, the on-chip DMA engine shows limited scalability on multi-channels, while using fewer channels incurs serious cross-app interferences. However, real-world applications have diverse performance goals, among them, some are latency-critical (`L-apps`) promised with service-level objectives (SLOs), while others are bandwidth-oriented (`B-apps`) with relaxed latency demands but higher bandwidth requirements for bulk data copies. Therefore, simply colocating these applications can lead to either SLO violations or device bandwidth underutilization. A feasible approach is regulating the bandwidth consumed by `B-apps` when `L-apps`' performance goals are unmet. Memory bandwidth regulation has been adopted many times in past work by assigning CPU quotas [25, 36], instrumenting software delays [73], or using Intel's memory bandwidth allocation (MBA) [75]. However, all these approaches are designed for applications that use `load/store` instructions to access memory, which is inapplicable for EasyIO since memory operations are offloaded to the DMA engine.

4 EasyIO Design

In this section, we introduce EasyIO and its key design techniques – *orderless file operation* (§ 4.2) and *two-level locking* (§ 4.3) that accelerate read and write operations and eliminate deadlock problems, and the *channel manager* (§ 4.4) that carefully mediates the interaction between DMA requests and channels to fulfill applications' diverse performance goals.

4.1 Overview

EasyIO adopts a novel combination of uthread scheduling and asynchronous I/Os to improve CPU efficiency. Figure 5 depicts the overall architecture of EasyIO. Applications run

inside normal processes, which spawn uthreads to execute application tasks; these uthreads are multiplexed on physical cores and scheduled by the userspace scheduling runtime. The slow memory is managed by a filesystem (e.g., NOVA in the kernel), which provides standard `read()` and `write()` syscalls to applications but with the asynchronous I/O semantics enabled. When an application issues an asynchronous I/O, the corresponding syscall is intercepted by the scheduling runtime first (❶ in the figure); this allows the runtime to make scheduling decisions upon each asynchronous I/O. Then, the runtime invokes the actual syscall (❷) and the filesystem processes it. The filesystem offloads memory copy operations of data pages to the DMA engine and returns back to the userspace runtime immediately before the DMA request completes (❸). In the meantime, the DMA engine performs data copy and updates the completion buffer to claim completion (❹❺). Once returning from an asynchronous syscall, the runtime polls completion from previously posted syscalls (❹) and performs a uthread switch to execute the uthread whose syscall has been finished (❺).

In this way, asynchronous I/Os and computation tasks from different uthreads are interleaved on the same core, while the complexity of fine-grained scheduling is hidden from applications to the runtime.

4.2 Orderless File Operation

In a `write()` operation, metadata updates are strictly ordered after data page updates (see the upper left of Figure 6), forcing EasyIO to issue multiple syscalls to accomplish an asynchronous I/O. In EasyIO, however, we observe a redundancy between the completion buffer and file metadata – both of them can describe the completion of writing data pages. With this extra information, we decouple the strict order between metadata and data and allow them to be processed in parallel (lower left of Figure 6).

In the DMA engine, each channel owns a completion buffer, which stores a 64-bit integer to always point to the most recently finished DMA descriptor in the hardware queue (upper right of Figure 6). In EasyIO, we place these completion buffers in a persistent region with their starting addresses recorded in advance. This allows the information stored in completion buffers to survive across system/power failures. Meanwhile, when updating the metadata for a `write()` operation, we also encapsulate the channel ID and the address of the DMA descriptor for this write in the block mapping as well (lower right of Figure 6). In this way, the metadata update can be committed before the DMA request finishes since the completion of data updates is described indirectly – we can determine whether a write operation is finished by comparing the embedded information in the block mapping with that in the completion buffer.

An essential prerequisite for the above mechanism to work properly lies in that the `ADDR` in the completion buffer should be increased monotonically. However, the hardware queue

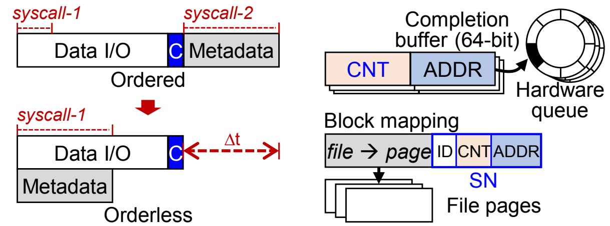

Figure 6. The design of orderless file operation. *File pages are read or written asynchronously using the DMA engine; metadata is updated using* `memcpy`*. The letter* `'C'` *in blue boxes indicates the completion of DMA.*

is essentially a ring buffer with a limited size, and the address stored in the completion buffer must wrap around. To handle wraparounds, we add an extra 64-bit counter alongside each completion buffer, whose value is added by one for each wraparound of the hardware queue. Hence, the counter (`CNT`) together with the channel ID (`ID`) and completion buffer (`ADDR`) form a sequence number (`SN`, see the right half of Figure 6), which is monotonically increased whenever a DMA request completes. Wraparounds of the counter are actually impossible (every 38 billion years) and are beyond our consideration.

With the above design, we no longer require a strict order between data and metadata. To process a `write()` operation, EasyIO first issues a DMA request for copying file data from the user buffer to slow memory; meanwhile, the file metadata is updated by embedding the `SN` related to the descriptor of this DMA request in the block mapping as well. Once the metadata is durably committed, EasyIO returns back to userspace for further scheduling. `read()` operations are processed similarly, despite that they do not modify block mappings but only timestamps. To allow EasyIO to check completion for read and write operations directly in userspace, we export the completion buffers' address space to userspace in a *readonly* mode.

If the system crashes in the middle of a write operation, we can determine whether the operation is finished and correctly recover the file by comparing `SN`s. First, we check for uncommitted file metadata (e.g., the tail pointer does not point to the end of the log in NOVA) and discard them accordingly. For those most recently committed file metadata, we further compare their `SN`s in the block mapping with that stored in the completion buffer. A block mapping update is considered valid only if the `SN` in the completion buffer is equal to or greater than that in the block mapping; otherwise, the committed metadata is discarded.

By default, we disable Intel DDIO (Data Direct I/O Technology[3]) to ensure that the data is durably written to the device by the DMA engine. This might be harmful to systems that heavily rely on DDIO to improve performance (e.g., networked storage systems [26, 46, 52]); On platforms with Intel

[3]DDIO allows devices to interact directly with the processor cache.

6

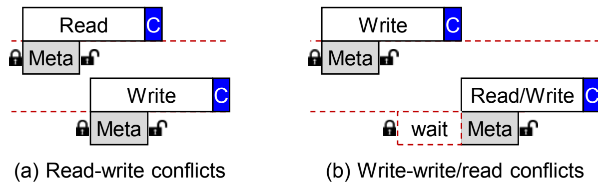

Figure 7. The design of two-level locking.

eADR [41], the data persistency is guranteed even with DDIO is enabled, however, due to the out-of-order cache flush problem, the write performance is significantly diminished [69]. Fortunately, this issue is expected to be addressed in the upcoming platform where DSA can enable and disable DDIO on a per-request basis.

4.3 Two-Level Locking

With the orderless file operation design (§ 4.2), EasyIO returns back to the userspace runtime when the metadata update has been committed but the I/O is unfinished. Unlocking the file immediately after the metadata update eliminates the risk of deadlocks (§3), since both locking and unlocking appear within the same syscall. However, early unlock raises consistency issues since this asynchronous I/O is unfinished yet and other concurrent operations may access inconsistent data. Again, we rely on the completion buffer to allow EasyIO to safely unlock files in advance and thus introduce the *two-level locking* mechanism.

When a `read()` or `write()` begins, it grabs the file lock as usual (*level-1 lock*) and releases the lock after metadata updates are committed. Due to the early release of the lock, we need to further regulate potential conflicts (*level-2 lock*); we treat read-write conflicts and write-write/read conflicts differently. `read()` operation does not modify block mapping and file data, so later writes on the same file can start immediately once the `read()` releases the lock, despite that its data I/O has not been finished (see Figure 7a). Besides, we require `write()` operations to use CoW to update file data, where old data pages are never modified in place; in this way, previously unfinished reads are guaranteed to not read dirty data. For a write-read/write conflict where an earlier `write()` has released the lock but its data copy is unfinished, the current read or write operation must block until it actually completes; otherwise, it might access inconsistent or obsolete data (see Figure 7b). Similar to the recovery logic in § 4.2, EasyIO determines whether a previous `write()` is completed by comparing the `SN` within the completion buffer with that of the most recently updated block mapping of this file. The way to find the most recently updated block mapping varies depending on how a filesystem is implemented: in NOVA, the newest updates appear at the tail of the metadata log; in extent-based filesystems (e.g., Ext4 [8]), we can store `SN` in the reserved field of the inode structure.

4.4 Channel Manager

We introduce the channel manager to meet applications' diverse performance goals. First, the channel manager provides *quality-of-service* guarantees for `L-apps` while maintaining high device bandwidth utilization for `B-apps`. Second, the channel manager enables *selective offloading* to maximize the device bandwidth while reducing CPU consumption.

Quality-of-service. The channel manager provides quality-of-service guarantees for `L-apps` by separating and throttling `B-apps`' bandwidth consumption. Recall that sharing DMA channels can lead to severe head-of-line blocking in the hardware queue (§2.2), we first assign `L-` and `B-apps` with separate DMA channels. Specifically, all `B-apps` share a single DMA channel, while `L-apps` steer DMA requests to up to 4 channels to balance the performance of writes and reads according to our empirical study.

We throttle `B-apps`' bandwidth consumption by manipulating the channel command register (`CHANCMD`) of I/OAT, which can suspend and resume a channel flexibly at a low cost (74 ns). By controlling the working time of a DMA channel, we can control its bandwidth consumption within a threshold of any value. For instance, given a target bandwidth threshold, the channel manager monitors the total amount of data copied by this channel within a period of time (i.e., epoch); if it reaches its limit for this epoch, the channel is suspended until the next epoch. To avoid an `L-app`'s SLO from being violated, we must suspend and resume `B-apps`' DMA channel at a fine-grained timescale (e.g., μs-scale) to prevent `B-apps` from occupying the device bandwidth for too long. However, we observe that when a DMA channel is suspended, the currently running DMA request is either executed to completion or restarted upon resumption, depending on how far the request has been processed. In the latter case, frequently suspending channels can cause the same I/O to be executed repeatedly if the I/O itself is large. We address this issue by splitting bulk I/Os of `B-apps` into small-sized ones (e.g., 64KB) before posting them to the DMA channel.

EasyIO employs an approach similar to Caladan [36] to guarantee `L-apps`' quality-of-service (see Listing 1). In each epoch, EasyIO monitors whenever an `L-app`'s SLO (e.g., latency) is met. If an SLO violation happens, the channel manager throttles `B-apps` by decreasing its bandwidth threshold; otherwise, the bandwidth threshold is increased instead.

```
1  while true:
2     min = MAX;
3     // Step-1: monitor L-apps' SLO
4     for L in L-apps:
5        tmp = (L.target - L.latency) / L.target;
6        min = (min < tmp) ? min : tmp;
7     // Step-2: conduct throttling decisions
8     if min < 0:
9        // Throttle down B-apps
10       B_APP_BW_LIMIT -= Delta;
11    else if min > threshold:
```

7

```
12          // Throttle up B-apps
13          B_APP_BW_LIMIT += Delta;
14      sleep(epoch);
```

Listing 1. Bandwidth regulation in pseudo-code. *Delta is a hyper-parameter concluded from experiments*

Selective offloading. Based on our empirical study in § 2.2, we offload memory copy operations to the DMA engine selectively due to its uneven performance advantages over memcpy across I/O types and I/O sizes. For I/O sizes smaller than 4KB, we choose to use *memcpy* directly and the reason is two-fold: first, the DMA engine is less efficient with small I/Os even with batch submission enabled; second, the time it takes to complete a small-sized I/O is very short (less than 1 μs), where the DMA request completion can happen simultaneously or even earlier than the CPU core returns back to userspace, leaving no CPU cycles to be harvested.

For read operations, the DMA engine saves CPU cycles but achieves much lower performance than memcpy. Hence, EasyIO uses both the DMA engine and memcpy for read operations, and adopts a simple admission control policy to balance I/Os dynamically between them (see Listing 2). Specifically, we allow a CPU core to issue read requests to the DMA engine only if its size is greater than 4KB and there is a DMA channel with a queue depth of less than 2; otherwise, CPU-involved memcpy is used for reading data. In this way, We make full use of the asynchronous nature of the DMA engine to save CPU resources when the read concurrency is low; under high concurrency, extra read requests are shunted to memcpy to realize higher total read bandwidth.

```
1  if req.size > 4K:
2      for C in DMA-Channels:
3          if C.q_deps < 2:
4              C.enqueue(req);
5              return;
6  // memcpy if size <= 4KB or no available channels
7  memcpy(req.des, req.src, req.size);
```

Listing 2. Read admission control in pseudo-code.

5 Implementation

Applying EasyIO to existing filesystems. The techniques introduced by EasyIO is orthogonal to how a filesystem formats data, and only require small modifications being made in data movements, concurrency control and recovery. Take NOVA [74] for an example. NOVA is a highly efficient and scalable persistent memory filesystem that organizes file metadata with a log structure and updates data pages using CoW. We modify NOVA in the following aspects to work with EasyIO (involving less than 50 lines of code changes). First, we modify the read and write code path to interact with the channel manager for data movements. When a file range is discontiguous on physical memory, NOVA issues multiple memcpys for each range; in EasyIO, we generate one DMA request per range and batch submit them to the DMA engine. We also add an extra SN field in the log entry (a basic file metadata structure) to record the corresponding DMA descriptor in the write code path. Second, we modify the locking logic of NOVA to use the two-level locking mechanism. Third, we supplement the recovery logic to discard those committed but invalid metadata by comparing SNs.

Userspace thread scheduling. Caladan [36] is a famous core scheduling framework that can balance load across cores within an application by stealing uthreads from busy cores, and reallocate cores between applications in reaction to load changes. We modify Caladan [36] to perfectly interact with asynchronous I/Os by adding two new functionalities: (i) the runtime needs to intercept synchronous syscall such as read and write; (ii) the runtime needs to poll for the completion of former issued DMAs and make context switch decisions based on this to guarantee correctness. In detail, we make the following modifications: First, the runtime intercepts filesystem syscalls transparently via the LD_PRELOAD environment variable. Second, as mentioned in Section 4, EasyIO exposes completion buffers to userspace in read-only mode. Therefore, the runtime can easily poll the completion by scanning these buffers to detect any memory changes. Originally, Caladan performs context switch in three conditions: *a.* when a uthread voluntarily releases the core (i.e., thread_yield()); *b.* when a uthread exits actively (i.e., thread_exit()); *c.* when the system is under imbalance load (i.e., work stealing). Slightly different in EasyIO, we perform the thread_yield() every time when returning from the kernel after sending an asynchronous I/O to transfer the execution of the core to the runtime. At this point, the runtime polls the completion directly, and right after that, the runtime performs context switch if necessary. By polling for completion directly, EasyIO avoids waste of CPU for dedicated polling threads. When doing context switch, EasyIO chooses the next runnable uthread if the completion of the uthread's previously issued syscall arrives or the uthread is at the head of the work queue. For uthreads whose I/Os have already been finished but are not resumed due to the current core is occupied by a long-term task, Caladan schedules other idle cores to steal them for execution. To guarantee correctness, EasyIO ensures that a uthread will not be switched to execution with former issued DMAs unfinished.

Compatibility with DSA. The recently introduced DSA is expected to deliver higher performance than I/OAT [48]. In this part, we discuss EasyIO's compatibility with DSA and how EasyIO can be easily ported to use DSA. Different from I/OAT's abstraction of channels, DSA consists of a set of configurable interfaces for communication with the host; among them, work queues (WQs) hold submitted work descriptors from the software, and processing engines (PEs) fetch descriptors from WQs and process them. Group is a basic unit for DSA (similar to I/OAT's channel), which can be configured with an arbitrary number of WQs and PEs, and an arbiter enforces fairness control between these PEs and WQs.

The completion buffer-centric design in EasyIO (§ 4.2 and 4.3) can be equally supported by DSA since it also uses completion buffers to record the completion of DMA requests. For bandwidth regulation, DSA provides similar features that can be used by EasyIO to disable and enable WQs dynamically using the command register. We can also further improve EasyIO leveraging DSA's arbiter. Currently, `L-apps` in EasyIO share a limited number of DMA channels, which incurs head-of-line blocking in the hardware queue when the number of `L-apps` increases. With DSA, we can create a WQ for each `L-app` and assign these WQs with different priorities using the arbiter, which adjusts the frequency at which descriptors are dispatched from WQs to PEs, and thus enables priority-based I/O scheduling. DSA is also expected to provide better performance for small IOs and read operations, enabling EasyIO to divert more I/O traffic to the DMA engine to free more CPU cycles. We leave this as our future work.

6 Evaluation

In our evaluation, we try to answer the following questions:

- Does EasyIO achieve the goal of reducing CPU resource consumption when compared with conventional synchronous filesystems?
- How does each individual technique in EasyIO help with achieving the above goals?
- How does EasyIO perform under real-world applications?

6.1 Experimental Setup

Testbed. Our experimental testbed is equipped with 2× Intel Xeon Gold 6240M CPUs (36 physical cores) and 192 GB DDR4 DRAM. Each CPU has 8 I/OAT channels, and the server is installed with 6 Intel Optane `DCPMMs` (256GB per DIMM, 1.5 TB in total) to work as slow memory. Unless otherwise stated, we perform all experiments on all Optane DCPMMs residing on both NUMA sides, whose read bandwidth peaks at 37.6 GB/s and the write bandwidth peaks at 13.2 GB/s. The server is installed with Ubuntu 18.04 and the kernel version is Linux 5.1, the newest kernel version supported by NOVA.

Compared filesystems. We evaluate EasyIO against three representative PM-aware filesystems and all of them provide synchronous interfaces. Among these filesystems, NOVA [74] can only use the PM spaces from one NUMA node, and we extend it to support using PM spaces from all NUMA nodes. `Fastmove` [69] is a PM filesystem that extensively uses DMA to accelerate file operations. However, it fails to run our benchmark and applications, so we implement a similar version named NOVA-DMA by replacing the `memcpy`s in the read and write code path with DMA-offloaded ones. OdinFS [76] is a NUMA-aware scalable filesystem that adopts a delegation mechanism for data movement, where background threads access PM on behalf of the application to automatically parallelize the accesses across NUMA nodes. The open-sourced

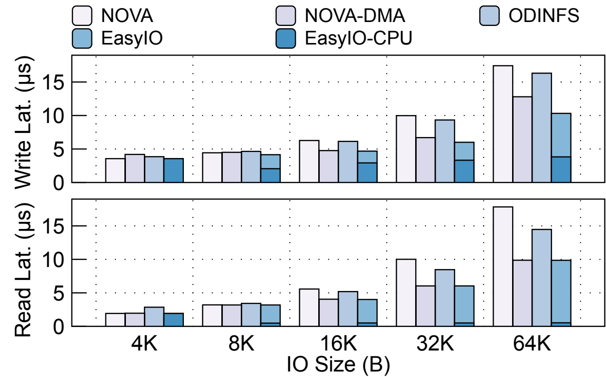

Figure 8. Latency comparison of write and read operations with varying IO sizes. EasyIO-CPU *indicates the time the CPU spent performing the operation in* EasyIO.

OdinFS runs on Linux 5.13 so we switch to this version for testing. By default, OdinFS reserves 12 physical cores per NUMA node to run background threads, and we keep the same configuration for evaluation.

6.2 Overall Performance

We use FxMark [58] to measure the overall performance of EasyIO against the compared filesystems in terms of both latency and throughput. FxMark provides 19 micro-benchmarks with tunable parameters to change data types, access modes, operation types, and sharing levels. In this paper, we focus on the `read()` and `write()` operations.

End-to-end latency with a single thread. Figure 8 shows the latency of write and read operations across all evaluated filesystems using 1 working thread. In EasyIO, we only run one uthread on a physical core and we let it busy-poll the DMA completion after returning from the kernel. We make the following three observations.

First, EasyIO achieves the lowest latency for both write and read operations. For example, the read latency of EasyIO is 22.49% lower than that of NOVA on average, while the write latency is 22.43% lower. For read operations, the latency reduction mainly comes from the benefits of DMA offloading; due to the same reason, NOVA-DMA also shows similar latencies as that of EasyIO. For write operations, EasyIO allows the metadata and data to be performed in parallel (i.e., orderless file operation), which further shortens the operation execution time; hence, EasyIO is even faster than NOVA-DMA. OdinFS also shows better latency than NOVA, especially for large I/Os, since it splits large I/Os into small ones and allows them to be performed in parallel by background threads.

Second, the latency reduction of EasyIO is more significant for large I/Os. Specifically, EasyIO exhibits up to 41% lower write latency than NOVA with the 64KB I/O size, while their latencies are almost similar for small I/Os. The reason is twofold: ① the latency for writing a small I/O is dominated by the metadata parts (indexing, updating, and committing), so

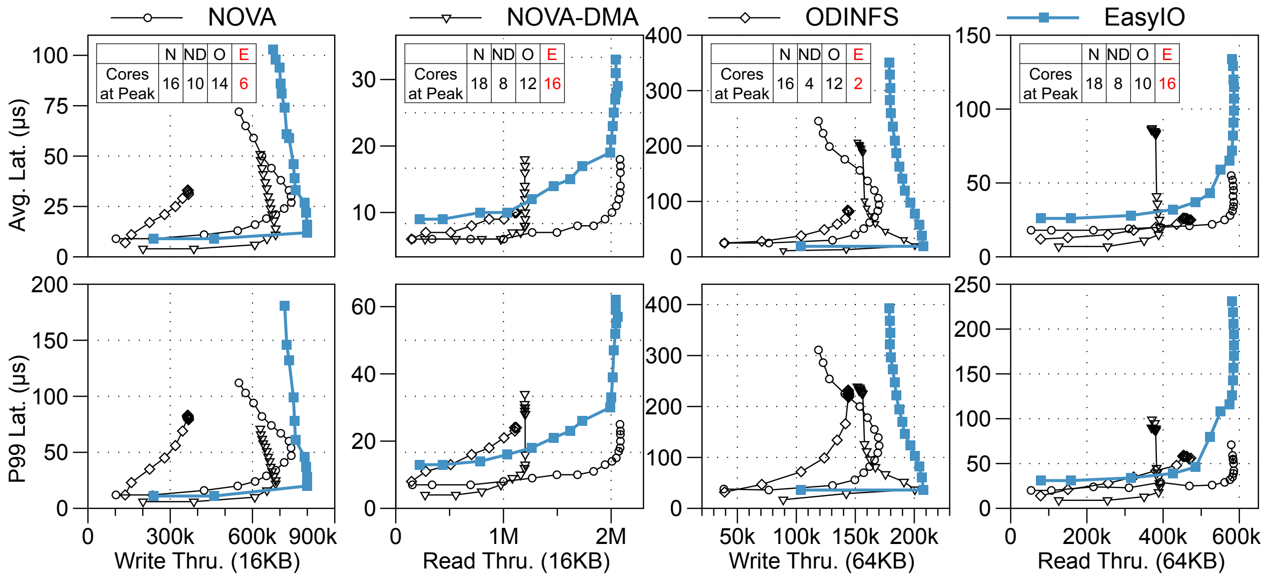

Figure 9. Throughput vs. latency by increasing the number of cores. *Tables embedded in the figure shows the minimum number of cores required to achieve a peak throughput. Letters* `N`*,* `ND`*,* `O`*, and* `E` *in tables indicate NOVA, NOVA-DMA,* ODINFS *and* EASYIO*, respectively.*

the benefits of offloading data I/O are limited; and ② EASYIO directly uses `memcpy` for 4KB I/Os.

Third, the time spent by the CPU performing the operation in EASYIO only occupies a small fraction (37% and 5% at 64KB) for both read and write operations. This indicates that nearly 63% and 95% of CPU time can be harvested from write and read operations respectively to run application tasks. Instead, NOVA, NOVA-DMA, and ODINFS require the involvement of the CPU all the time to accomplish file operations due to their synchronous interfaces.

Throughput vs. latency. We measure the throughput achieved by all evaluated filesystems and the associated average and 99th percentile (P99) latencies as we increase the number of working threads. Here we use FxMark's `DWAL` and `DRBL` workloads, which allow multiple working threads to write (read) to (from) their private files. For EASYIO, FxMark runs inside Caladan and each uthread runs a worker (the number of created uthreads is two times of physical cores). The top half of Figure 9 shows the average latency and the bottom half shows the P99 latency as the throughput increases. The tables embedded in Figure 9 show the minimum number of cores required for each filesystem to achieve a peak throughput. We make the following three observations:

First, EASYIO uses much fewer cores to achieve a peak throughput for write operations. For example, EASYIO requires 6 cores to peak in write throughput with 16KB I/Os; with larger I/Os (e.g., 64KBs), EASYIO only uses 2 cores. Compared to NOVA, EASYIO saves 63% and 88% CPU cores, respectively, to achieve similar performance. With EASYIO's asynchronous I/O interfaces, uthreads in Caladan do not need to block for I/O completions, instead, they are scheduled to run other uthreads, allowing more asynchronous I/O to be issued, and thus save most CPU resources. EASYIO also saves CPU resources for read operations but the benefit is rather limited. This is because the DMA engine only provides a limited bandwidth for read operations, forcing a large part of the read traffic to be moved by `memcpy` in EASYIO. In this way, only a small fraction of CPU cycles are harvested. Still, EASYIO achieves an 11% reduction in CPU consumption than NOVA. We also notice that the read throughput of NOVA-DMA and ODINFS peaks at fewer cores than EASYIO, but their achieved peak throughput is much lower than that of EASYIO. NOVA-DMA solely uses DMA to copy data and its peak throughput is only less than half of what EASYIO achieves. ODINFS needs to reserve many cores to run background threads, so we can only use up to 12 cores to run worker threads of FxMark with ODINFS, which is not enough to realize a peak throughput.

Second, EASYIO achieves lower write latency but higher read latency than the compared filesystems. For write operations, EASYIO can restrict its average latency below $16\mu s$ and $19\mu s$, respectively, for the 16KB and 64KB I/O sizes as the throughput peaks, and the corresponding P99 latency is as low as $28\mu s$ and $36\mu s$; this is 41% and 71% lower than that of NOVA on average. The main reason lies in that EASYIO requires only several cores (2 - 6 cores) to achieve a peak write throughput, where the head-of-line blocking in the hardware queue and the device-level bandwidth contention are not significant. For the same reason, NOVA-DMA also achieves better write latency. EASYIO exhibits higher read latency; for example, given a target throughput running at 2Mops with the 16KB I/O size, EASYIO's average and P99 latencies are

90% and 114% higher than that of NOVA. As mentioned before, EasyIO uses both asynchronous DMA and synchronous `memcpy` to process read traffic; when they are interleaved together, asynchronous I/Os will be further delayed by synchronous I/Os since the running core will continue executing the same uthread when the synchronous I/O finishes.

Third, EasyIO achieves the highest peak write throughput among the compared filesystems (13% higher than NOVA on average) and its throughput only declines slightly as we further increase the number of working threads. The write throughput of both NOVA and NOVA-DMA decreases sharply under higher concurrency but with different reasons behind it. Optane DCPMM has poor write scalability [27, 76] and is responsible for NOVA's performance decline. NOVA-DMA uses all available DMA channels, and our empirical study in §2.2 shows that using more channels is harmful to performance scalability. Instead, EasyIO uses fewer channels, so its write throughput only drops slightly.

6.3 Real-World Applications

In this section, we evaluate how EasyIO performs with the following eigth real-world applications, and Table 1 shows the read size, write size and read/write ratio of them.

- **Snappy** [39] is a compression/decompression library. We let working (u)threads read compressed data from pre-built files, decompress it, and write it into new files.
- **JPGDecoder**[14] reads pictures from files, decodes them into RGB888 format, and writes them into new files.
- **AES** is a cryptographic algorithm contained in Crypto [5]. We let working (u)threads encrypt files using AES into ciphertext and write them to new files.
- **Grep** [10] is a commonly used utility for searching plain-text data sets for lines that match a regular expression. We let each worker (u)thread read data from a separate file into a buffer and do string matching for each line in that buffer.
- **KNN** [15] (k-nearest neighbors) relies on the k-d tree, a binary tree whose tree node is a k-dimensional vertex, to make classifications or predictions of an individual data point. We let working (u)threads read sample data from files and search for each sample in a pre-built k-d tree.

Application	Avg. Read Size	Avg. Write Size	R/W Ratio
Snappy	910KB	1.9MB	1:1
JPGDecoder	343KB	6.3MB	1:1
AES	64KB	64KB	1:1
Grep	2MB	0KB	1:0
KNN	1MB	0KB	1:0
BFS	1MB	0KB	1:0
Fileserver	1MB	1040KB	1:2
Webserver	256K	16KB	10:1

Table 1. Configuration of real-world applications.

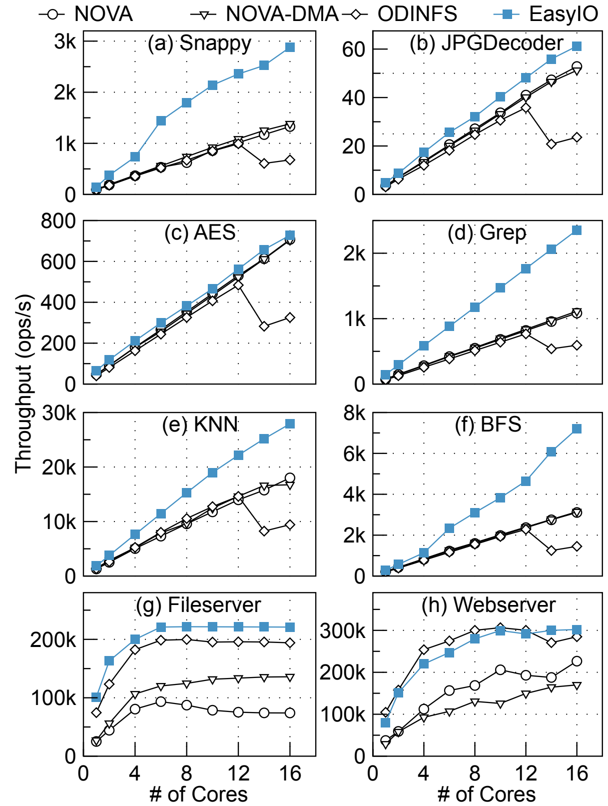

Figure 10. Throughput of real-world applications.

- **BFS** [1] (Breadth First Search) reads serialized graph data from files, builds the graph in memory, and runs the breadth-first search from a given vertex.
- **Fileserver** is a workload from Filebench [7], it performs *create*, *write*, *read*, *stat*, *delete* operations on a set of files.
- **Webserver** is a workload from Filebench, it performs a large number of *read* operations on a set of files and then appends to a single log file.

As shown in Figure 10, for I/O intensive or I/O-computation balanced applications, such as Snappy, Grep, KNN, and BFS, EasyIO achieves 2.1×, 2.1×, 1.5×, and 2.3× higher throughput than NOVA, respectively, as we increase the number of worker threads. EasyIO enables asynchronous I/Os and allows computation to be interleaved with I/Os, thus using CPU more efficiently given the same number of cores. In other synchronous filesystems, the CPU cores are fully involved in both I/Os and computation, so they deliver much lower throughput than EasyIO. JPGDecoder and AES are both computation-dominated, where decoding and encrypting data occupy most CPU cycles. Due to this reason, EasyIO only achieves slightly higher throughput than other synchronous filesystems. NOVA-DMA also uses DMA for data copy, but it only exposes synchronous interfaces to applications, leaving no CPU cycles to be harvested. For Fileserver, EasyIO shows 2.3×, 1.6× and 1.1× higher throughput than NOVA,

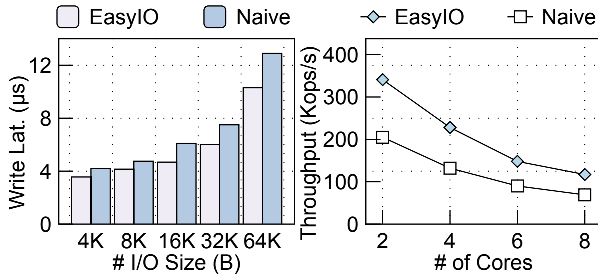

Figure 11. Effectiveness of orderless file operation and two-level locking. *The left half shows the write latency with different I/O sizes; the right half shows the throughput with lock conflicts.*

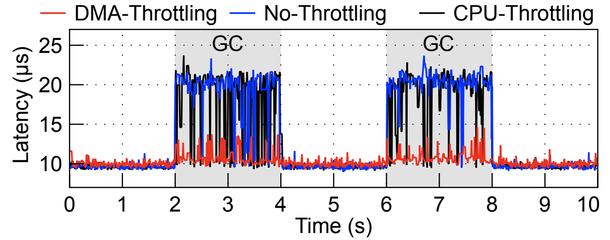

Figure 12. Effectiveness of bandwidth throttling. *We colocate a Web server (64KB I/O size) with a garbage collector (issues 2MB bulk I/O periodically), and report the latency of the Web server.*

NOVA-DMA and OdinFS respectively. NOVA achieves the lowest throughput because, for a write operation, NOVA uses one core to write data to DCPMM with memcpy, which is less efficient than DMA and multiple cores delegation as in OdinFS. Under high contention(i.e., Webserver in Figure 10(h), however, EasyIO shows limited opportunity to harvest CPU resources efficiently, therefore achieves lower throughput than OdinFS. OdinFS's throughput starts to decline beyond 12 cores since we only have 12 cores left due to the reservation of cores for background threads; for this reason, we only do experiments within 16 cores.

6.4 Effects of Individual Techniques

In this section, we measure the effectiveness of each individual technique. In order to highlight the effectiveness of the *Orderless File Operation* and the *Two-Level Locking*, we implement a naive version of EasyIO (namely Naive), which processes read and write operations with the data and metadata updating strictly ordered into separate syscalls.

6.4.1 Effects of orderless file operation. The left part of Figure 11 presents the latency of EasyIO and Naive for write operations with different I/O sizes. We observe that the latency of EasyIO is 18% lower than that of Naive on average. This is because EasyIO allows data and metadata to be updated in parallel with fewer syscalls (*Orderless File Operation*). Moreover, the latency improvements over Naive become much more obvious as the I/O size grows. The reason is that for smaller I/Os, updating metadata becomes dominant compared to data copy in DMA, which diminishes the effect of processing data and metadata in parallel.

6.4.2 Effects of two-level locking. Then we measure the throughput achieved by EasyIO and Naive with high contention on file locks to demonstrate the effects of *two-level locking*. To avoid deadlocks of Naive, we disable work stealing in Caladan and create two different uthreads on each physical core, and let one uthread run the FxMark `DWOM` workload, which writes to a shared file, and the other performs scientific computation that never issues I/Os.

As shown in the right part of Figure 11, both EasyIO and Naive deliver less throughput as the number of cores increases. The reason lies in that more cores incur higher lock contention overhead (i.e. more writers race for the same lock). Even so, EasyIO achieves a throughput that is 66% higher than Naive does with 2 cores. This is because Naive holds the lock all the time for a write operation, and intermediate scheduling within a write operation further prolongs the critical section, making later writes to be delayed. The two-level locking in EasyIO allows the write lock to be released in advance without delaying other conflicting operations.

6.4.3 Effects of bandwidth throttling. To evaluate the effectiveness of the *channel manager*, we colocate two applications, where a foreground application is a Web server (`L-app`), which reads HTML files in response to client requests; the background application is a garbage collector (`B-app`) that conducts bulk data movement (2MB) periodically. The client requests of the Web server follow a Poisson arrival distribution. As shown in Figure 12, we implement No-Throttling and CPU-Throttling for comparison. In No-Throttling, colocated applications run with EasyIO without throttling their bandwidth. In CPU-Throttling, EasyIO throttles the bandwidth consumed by the garbage collector by assigning it with fewer CPU cycles using the policy of Caladan. In both CPU-Throttling and DMA-Throttling, we regulate the bandwidth consumed by the garbage collector to be below 2GB/s (1/3rd of the hardware bandwidth).

In both CPU-Throttling and No-Throttling, the latency of Web server spikes immediately as GC starts, and the maximum latency increases up to 20μs, which is 2.5× higher than that when the GC instance is idle. On the contrary, DMA-Throttling presents better control over the latency of the Web server – its maximum latency is merely 13.7μs, which is 40% lower than that in CPU-Throttling and No-Throttling. CPU-Throttling fails to regulate bandwidth since EasyIO does not rely on `load/store` to move data. Hence, even if the garbage collector is assigned with fewer CPU cycles, it still can occupy the memory bandwidth using the DMA engine.

6.5 Crash Consistency with CrashMonkey

Workloads	Description	Total Crash Points	Total Passed
create_delete	create, write, remove on regular files	1000	1000
generic_056	create, write, link on regular files	1000	1000
generic_090	write, append, link on regular files	1000	1000
generic_322	create, write, rename on regular files	1000	1000

Table 2. Crash consistency test results with CrashMonkey.

To evaluate if EasyIO recovers correctly when power failures happen, we use a black-box testing tool, CrashMonkey [59]. CrashMonkey automatically generates workloads and checks both data and metadata to determine the correctness of recovery. We choose 4 workloads which contain the *write()* operations to accurately test the consistency of EasyIO and performs 1000 tests for each workload. These workloads cover several error-prone syscalls including *create()*, *write()*, *delete()* and *rename()* etc. Table 2 shows the results. EasyIO guarantees crash consistency for the following reasons: (i) by recording SN in block mappings and incorporating with CoW, EasyIO can restore the filesystem to a consistent state by discarding block mappings (the log in NOVA) with unfinished DMAs; (ii) with two-level locking, EasyIO preserve the concurrency consistency; (iii) the runtime guarantees that a uthread will not be switched to execute with unfinished DMAs. Therefore, EasyIO passes all tests.

6.6 Summary

From the experimental results, we conclude that EasyIO shows limited benefit under three scenarios: first, under high contention workloads (e.g., Webserver in Figure 10(h)), EasyIO is less efficiency because the CPU is wasted on file lock; second, for I/O size smaller than 4KB, EasyIO is less efficiency because the I/OAT copies data slower than CPU; third, for read operations, EasyIO introduces higher latencies because I/OAT is less efficient in read. However, as mentioned before, DSA delivers higher performance than I/OAT, and we expect that DSA can further expand EasyIO's benefit.

7 Related Work

DMA-accelerated storage systems. Many recent work used the on-chip DMA to accelerate data movement operations in storage systems [21, 69] and network processing [37, 71]. Among them, Assise[21] uses I/OAT for cross-socket log eviction to NVMs; `Fastmove` extensively uses the on-chip DMA engine for accelerating `read()` and `write()` operations in a persistent memory filesystem. All above systems use I/OAT based on the fact that the on-chip DMA achieves better single-core performance than CPU-involved `memcpy`. However, our paper shows that, under high concurrency setups, I/OAT fails to provide higher peak bandwidth; instead, the core advantage of I/OAT lies in that it can release CPU resources for data movements. Therefore EasyIO adopts a fundamentally different approach (i.e., asynchronous IO), to harvest the otherwise wasted CPU cycles in data movements. Several studies also use I/OAT for improving CPU efficiency, including providing asynchronous memory copy interfaces [29, 70], or reducing CPU costs for background data migration tasks [63]; we share a similar goal as them but provide a more systematic approach for achieving high CPU utilization and transparency by introducing EasyIO.

Slow memory filesystems. A line of research work focuses on providing *file* abstraction to manage slow memory. Among them, single node filesystems are placed either in the kernel [30, 35, 43, 74], or in userspace [34, 49, 72], or in both [27, 44]; most of these systems focus on reducing the software overhead with diverse approaches to fully free up hardware bandwidth. In a distributed environment where memory devices are connected via the high-speed network (e.g., RDMA and CXL), recent work also propose to use *files* to manage the distributed shared memory space [19, 54, 68, 78] due to its familiar programming manner, mature software ecosystem, and full-fledged permission control. However, all the above filesystems provide synchronous interfaces; EasyIO is different from them by providing asynchronous I/Os to applications and thus achieves higher CPU utilization.

Asynchronous I/O. This is the opposite concept of synchronous IO; when an asynchronous I/O is issued, the caller does not immediately get the result, and the completion event is returned back via shared status, notifications, or callbacks. Asynchronous I/O is being extensively used to manage external storage and network devices. For example, in `libaio` [16], I/O request is submitted via `io_submit()` and completion information is acquired using `io_getevents()`; similar interfaces also appear in SPDK [3], DPDK [6], and RDMA verbs. With the asynchronous I/O interfaces, the CPU core can interleave application logic with I/Os to improve CPU efficiency. As far as we know, EasyIO is the first to exploit the asynchrony of slow memory storage systems.

Userspace core scheduling. In response to the 'killer microsecond' problem when processing microsecond-scale requests, existing systems improve CPU utilization by adopting user space core scheduling. Among them, Shenango [60] and Arachne [62] run applications inside uthreads and reallocates cores across applications at fine-grained time intervals; Caladan [36] steps further by introducing flexible and efficient policies to balance loads across cores and applications in response to load spikes. Another line of studies balance load across cores via work stealing [61], separating short and long tasks [32, 33], and fine-grained core preemption [45]. Some recent work also offloads the scheduler to SmartNICs to mitigate the CPU overhead [40, 53, 67]. These systems motivate us to use uthreads in EasyIO to improve CPU efficiency without compromising the software transparency.

8 Conclusion

We introduced EasyIO to enable asynchronous I/Os on filesystems designed for slow memories; with this, EasyIO mitigates the storage tax by harvesting the otherwise wasted CPU cycles to run applications with a novel combination of memory-level asynchronous I/O and userspace core scheduling. EasyIO does not provide higher peak performance than existing filesystems; instead, we aim at using fewer cores to achieve the same peak bandwidth. Currently, EasyIO is more performant for `write()` operations (saving up to 88% CPU resources) due to I/OAT's unbalanced performance between reads and writes; we believe that the benefits of EasyIO can be further extended with faster DMA engines (e.g., DSA).

9 Acknowledgements

We thank our shepherd Pramod Bhatotia and anonymous reviewers for their feedback and suggestions. This work is supported by the National Key R&D Program of China (Grant No. 2022YFB4500302), National Natural Science Foundation of China (Grant No. U22B2023, 62202255) and Huawei (TC20220808044). Youmin Chen is also supported by the National Postdoctoral Program for Innovative Talents (Grant No. BX2021154) and the China Postdoctoral Science Foundation (Grant No. 2021M701887).

References

[1] 2010. Breadth First Search or BFS for a Graph. https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/.

[2] 2019. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[3] 2023. Building ultra high-performance storage applications with the storage performance development kit. https://spdk.io/.

[4] 2023. Compute Express Link: The Breakthrough CPU-to-Device Interconnect CXL. https://www.computeexpresslink.org/.

[5] 2023. Crypto. https://github.com/openssl/openssl.

[6] 2023. Data Plane Development Kit. https://www.dpdk.org/.

[7] 2023. Filebench. https://github.com/filebench/filebench/.

[8] 2023. Fourth extended filesystem (ext4). https://www.kernel.org/doc/html/v4.19/filesystems/ext4/index.html.

[9] 2023. The Go Programming Language. https://golang.org/.

[10] 2023. Grep. https://github.com/PatricZhao/ParallelGrep.

[11] 2023. Intel Data Streaming Accelerator Architecture Specification. https://cdrdv2-public.intel.com/759709/353216-data-streaming-accelerator-user-guide-2.pdf.

[12] 2023. Intel I/O Acceleration Technology. https://www.intel.com/content/www/us/en/wireless-network/accel-technology.html.

[13] 2023. Intel Optane Memory - Responsive Memory, Accelerated Performance. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[14] 2023. JPGDecoder. https://github.com/fzyzwrj/TinyJPEGDecoder.

[15] 2023. Kdtree. https://github.com/jtsiomb/kdtree.

[16] 2023. Linux AIO. https://pypi.org/project/libaio/.

[17] 2023. Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/.

[18] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. 6, 11 (aug 2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[19] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novakovic, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. 2018. Remote Regions: A Simple Abstraction for Remote Memory. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference* (Boston, MA, USA) *(USENIX ATC '18)*. USENIX Association, USA, 775–787.

[20] Hiroyuki Akinaga and Hisashi Shima. 2010. Resistive random access memory (ReRAM) based on metal oxides. *Proc. IEEE* 98, 12 (2010), 2237–2251.

[21] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. 2020. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1011–1027. https://www.usenix.org/conference/osdi20/presentation/anderson

[22] Saman Barghi. [n. d.]. uThreads: Concurrent User Threads in C++(and C). https://github.com/samanbarghi/uThreads.

[23] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadat, and O. Mutlu. 2022. Hermes: Accelerating Long-Latency Load Requests via Perceptron-Based Off-Chip Load Prediction. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE Computer Society, Los Alamitos, CA, USA, 1–18. https://doi.org/10.1109/MICRO56248.2022.00015

[24] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. *ACM Trans. Parallel Comput.* 5, 3, Article 13 (jan 2019), 39 pages. https://doi.org/10.1145/3298989

[25] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 107–120. https://doi.org/10.1145/3297858.3304005

[26] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[27] Youmin Chen, Youyou Lu, Bohong Zhu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jiwu Shu. 2021. Scalable Persistent Memory File System with Kernel-Userspace Collaboration. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 81–95. https://www.usenix.org/conference/fast21/presentation/chen-youmin

[28] Youmin Chen, Jiwu Shu, Jiaxin Ou, and Youyou Lu. 2018. HiNFS: A Persistent Memory File System with Both Buffering and Direct-Access. *ACM Trans. Storage* 14, 1, Article 4 (apr 2018), 30 pages. https://doi.org/10.1145/3204454

[29] Zhenke Chen, Dingding Li, Zhiwen Wang, Hai Liu, and Yong Tang. 2021. RAMCI: a novel asynchronous memory copying mechanism based on I/OAT. *CCF Transactions on High Performance Computing* 3 (2021), 129–143.

[30] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[31] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (feb 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[32] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with PerséPhone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 621–637. https://doi.org/10.1145/3477132.3483571

[33] Diego Didona and Willy Zwaenepoel. 2019. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 79–94. https://www.usenix.org/conference/nsdi19/presentation/didona

[34] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-Space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 478–493. https://doi.org/10.1145/3341301.3359637

[35] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[36] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[37] Brice Goglin. 2008. Improving message passing over Ethernet with I/OAT copy offload in Open-MX. In *2008 IEEE International Conference on Cluster Computing*. 223–231. https://doi.org/10.1109/CLUSTR.2008.4663775

[38] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12)*. USENIX Association, USA, 17–30.

[39] Google. 2023. Snappy, a fast compressor/decompressor. https://github.com/google/snappy.

[40] Jack Tigar Humphries, Kostis Kaffes, David Mazières, and Christos Kozyrakis. 2019. Mind the gap: A case for informed request scheduling at the nic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*. 60–68.

[41] Intel. [n. d.]. What Is ADR? https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html.

[42] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[43] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: A Hugepage-Aware File System for Persistent Memory That Ages Gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 804–818. https://doi.org/10.1145/3477132.3483567

[44] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. https://doi.org/10.1145/3341301.3359631

[45] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for second-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[46] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 1–16. https://www.usenix.org/conference/nsdi19/presentation/kalia

[47] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 185–201. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia

[48] Reese Kuper, Ipoom Jeong, Yifan Yuan, Jiayu Hu, Ren Wang, Narayan Ranganathan, and Nam Sung Kim. 2023. A Quantitative Analysis and Guideline of Data Streaming Accelerator in Intel 4th Gen Xeon Scalable Processors. *arXiv preprint arXiv:2305.02480* (2023).

[49] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 460–477. https://doi.org/10.1145/3132747.3132770

[50] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 591–605. https://doi.org/10.1145/3373376.3378531

[51] Siyang Li, Youyou Lu, Jiwu Shu, Yang Hu, and Tao Li. 2017. LocoFS: A Loosely-Coupled Metadata Service for Distributed File Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 12 pages. https://doi.org/10.1145/3126908.3126928

[52] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[53] Jiaxin Lin, Adney Cardoza, Tarannum Khan, Yeonju Ro, Brent E. Stephens, Hassan Wassel, and Aditya Akella. 2023. RingLeader: Efficiently Offloading Intra-Server Orchestration to NICs. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1293–1308. https://www.usenix.org/conference/nsdi23/presentation/lin

[54] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an RDMA-enabled Distributed Persistent Memory File System. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 773–785. https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu

[55] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanan Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *Proc. VLDB Endow.* 11, 10 (jun 2018), 1303–1316. https://doi.org/10.14778/3231751.3231765

[56] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data* (Indianapolis, Indiana, USA) *(SIGMOD '10)*. Association for Computing Machinery, New York, NY, USA, 135–146. https://doi.org/10.1145/1807167.1807184

[57] Sarah McClure, Amy Ousterhout, Scott Shenker, and Sylvia Ratnasamy. 2022. Efficient Scheduling Policies for Microsecond-Scale Tasks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, Renton, WA, 1–18. https://www.usenix.org/conference/nsdi22/presentation/mcclure

[58] Changwoo Min, Sanidhya Kashyap, Steffen Maass, and Taesoo Kim. 2016. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 71–85. https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[59] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2018)*. USENIX Association, Carlsbad, CA.

[60] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[61] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[62] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 145–160. https://www.usenix.org/conference/osdi18/presentation/qin

[63] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 392–407. https://doi.org/10.1145/3477132.3483550

[64] Mohan Rokkam and Shyam Iyer. [n. d.]. Accelerating Intra-Host Data Movement with VMware PVRDMA on a Dell AMD PowerEdge Server. https://infohub.delltechnologies.com/p/accelerating-intra-host-data-movement-with-vmware-pvrdma-on-a-dell-amd-poweredge-server/.

[65] Zhenyuan Ruan, Seo Jin Park, Marcos K. Aguilera, Adam Belay, and Malte Schwarzkopf. 2023. Nu: Achieving Microsecond-Scale Resource Fungibility with Logical Processes. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 1409–1427. https://www.usenix.org/conference/nsdi23/presentation/ruan

[66] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[67] Hamed Seyedroudbari, Srikar Vanavasam, and Alexandros Daglis. 2023. Turbo: SmartNIC-enabled Dynamic Load Balancing of µs-scale RPCs. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 1045–1058. https://doi.org/10.1109/HPCA56546.2023.10071135

[68] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. 2017. Distributed Shared Persistent Memory. In *Proceedings of the 2017 Symposium on Cloud Computing* (Santa Clara, California) *(SoCC '17)*. Association for Computing Machinery, New York, NY, USA, 323–337. https://doi.org/10.1145/3127479.3128610

[69] Jingbo Su, Jiahao Li, Luofan Chen, Cheng Li, Kai Zhang, Liang Yang, and Yinlong Xu. 2023. Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 363–378. https://www.usenix.org/conference/fast23/presentation/su

[70] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. 2007. Designing Efficient Asynchronous Memory Operations Using Hardware Copy Engine: A Case Study with I/OAT. In *2007 IEEE International Parallel and Distributed Processing Symposium*. 1–8. https://doi.org/10.1109/IPDPS.2007.370479

[71] Karthikeyan Vaidyanathan and Dhabaleswar K. Panda. 2007. Benefits of I/O Acceleration Technology (I/OAT) in Clusters. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*. 220–229. https://doi.org/10.1109/ISPASS.2007.363752

[72] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 14, 14 pages. https://doi.org/10.1145/2592798.2592810

[73] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2022. NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 1–16. https://www.usenix.org/conference/fast22/presentation/wu

[74] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[75] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. 2022. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 199–216. https://www.usenix.org/conference/fast22/presentation/yi-mt2

[76] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 179–193. https://www.usenix.org/conference/osdi22/presentation/zhou-diyu

[77] Yang Zhou, Hassan M. G. Wassel, Sihang Liu, Jiaqi Gao, James Mickens, Minlan Yu, Chris Kennelly, Paul Turner, David E. Culler, Henry M. Levy, and Amin Vahdat. 2022. Carbink: Fault-Tolerant Far Memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 55–71. https://www.usenix.org/conference/osdi22/presentation/zhou-yang

[78] Bohong Zhu, Youmin Chen, Qing Wang, Youyou Lu, and Jiwu Shu. 2021. Octopus+: An RDMA-Enabled Distributed Persistent Memory File System. *ACM Trans. Storage* 17, 3, Article 19 (aug 2021), 25 pages. https://doi.org/10.1145/3448418