
# Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems

Jingbo Su, Jiahao Li, and Luofan Chen, *University of Science and Technology of China;* Cheng Li, *University of Science and Technology of China and Anhui Province Key Laboratory of High Performance Computing;* Kai Zhang and Liang Yang, *SmartX;* Sam H. Noh, *UNIST & Virginia Tech;* Yinlong Xu, *University of Science and Technology of China and Anhui Province Key Laboratory of High Performance Computing*

https://www.usenix.org/conference/fast23/presentation/su

This paper is included in the Proceedings of the 21st USENIX Conference on File and Storage Technologies.

February 21–23, 2023 • Santa Clara, CA, USA

978-1-939133-32-8

Open access to the Proceedings of the 21st USENIX Conference on File and Storage Technologies is sponsored by

# Revitalizing the Forgotten On-Chip DMA to Expedite Data Movement in NVM-based Storage Systems

Jingbo Su[†] Jiahao Li[†] Luofan Chen[†] Cheng Li[†ζ] Kai Zhang[§] Liang Yang[§]
Sam H. Noh[‡¶*] Yinlong Xu[†ζ]

[†]*University of Science and Technology of China* [‡]*UNIST* [¶] *Virginia Tech*
[§]*SmartX*[†] [ζ]*Anhui Province Key Laboratory of High Performance Computing*

## Abstract

Data-intensive applications executing on NVM-based storage systems experience serious bottlenecks when moving data between DRAM and NVM. We advocate for the use of the long-existing but recently neglected on-chip DMA to expedite data movement with three contributions. First, we explore new latency-oriented optimization directions, driven by a comprehensive DMA study, to design a high-performance DMA module, which significantly lowers the I/O size threshold to observe benefits. Second, we propose a new data movement engine, `Fastmove`, that coordinates the use of the DMA along with the CPU with judicious scheduling and load splitting such that the DMA's limitations are compensated, and the overall gains are maximized. Finally, with a general kernel-based design, simple APIs, and DAX file system integration, `Fastmove` allows applications to transparently exploit the DMA and its new features without code change. We run three data-intensive applications MySQL, GraphWalker, and Filebench atop `NOVA`, `ext4-DAX`, and `XFS-DAX`, with standard benchmarks like TPC-C, and popular graph algorithms like PageRank. Across single- and multi-socket settings, compared to the conventional CPU-only NVM accesses, `Fastmove` introduces to TPC-C with MySQL 1.13-2.16× speedups of peak throughput, reduces the average latency by 17.7-60.8%, and saves 37.1-68.9% CPU usage spent in data movement. It also shortens the execution time of graph algorithms with GraphWalker by 39.7-53.4%, and introduces 1.12-1.27× throughput speedups for Filebench.

## 1 Introduction

Emerging non-volatile memory (NVM) technologies such as STT-MRAM [45], PCM [40], ReRAM [6], and 3D-XPoint [15] offer byte-addressibility and comparable latency as DRAM but with substantially larger capacity. In addition, it provides data durability with orders of magnitude higher performance than prior durable devices like SSDs [55]. Recently, numerous studies have been proposed to combine faster, volatile DRAM, for caching, with slightly slower, denser NVM, for persisting data, in storage systems to revolutionize I/O performance of data-intensive applications with persistence demands [12].

In NVM-based storage systems, data are often moved between the two types of memories, due to DRAM cache fill-up, logging, or flushing. However, recent studies [28, 55] highlight that the DRAM-NVM data movement is not efficient, mainly because of their performance gaps in latency and bandwidth [13]. Additionally, we further notice that such data movement leads to heavy CPU consumption since NVM chips are attached to the memory bus, and their accesses must make use of the `load` and `store` instructions. Such negative performance effects worsen with multiple sockets, which modern high-end servers often provide, because of the negative NUMA impact [28]. This data movement bottleneck severely impairs the overall performance of I/O intensive applications and consequently, undermines the benefits brought by incorporating NVM.

To address this bottleneck, the slowness of NVM motivates us to re-think the usage of the on-chip DMAs that still come with the CPU but have deteriorated in use with the advent of fast storage devices. In this paper, we seek to transparently expedite data movement in NVM-based storage systems by (partially) offloading data movement to DMA to improve overall performance. However, while exploiting the on-chip DMA is a natural optimization, there are a few obstacles to incorporating it into NVM-based storage systems.

First, we need to handle more complex I/O patterns and have significantly different optimization goals than existing work [41, 54], which have already applied DMA as a minor technique to free CPU cycles of page migration in tiered DRAM-NVM systems. They handle I/Os that are always large, i.e., 2MB, and run in the background. However, NVM-based storage systems face I/Os with much smaller and variable sizes that are often on the critical path of the foreground user requests. Thus, our primary optimization goal is to shorten the execution time of DMA requests. Second, latency-critical optimization requires an in-depth understand-

[*]This work was done at UNIST.
[†]"SmartX" is also known as Beijing Zhiling Haina Technology Co., LTd.

ing of the strengths and limits of DMA, in conjunction with NVM and storage-facing I/Os, which is largely beyond existing studies [21].

To address the above challenges, first, we conduct a comprehensive study to understand the latency behaviors of using DMA for DRAM-NVM data movement on the Intel I/OAT and Optane PM combination, the only such pair in existence. This study suggests that the potential of DMA is heavily constrained by various factors, e.g., uneven advantages between reads and writes over the CPU, the non-negligible costs that grow with I/O size, bandwidth and concurrency limits, etc.

Second, we derive principles from the study to design `Fastmove`, a general data movement system that sits at the lower level of the software hierarchy. At the core, it includes a high-performance DMA module, which encapsulates the upper-level I/O requests into low-level hardware commands that comply with the workflows of data movement in NVM-based storage systems. We also maximize the benefits of DMA by introducing various optimizations such as batching the page pinning and descriptor submission activities for grouped DMA tasks and balancing and coordinating concurrent accesses to DMA channels. Furthermore, to compensate for the limitations of the stand-alone DMA solution such as the extra overhead and the concurrency and bandwidth constraints, we devise a lightweight `Scheduler` to prioritize bulk I/Os to go through DMA, while smaller I/Os are routed to the original CPU path. `Scheduler` additionally splits bulk read I/Os and balance loads between the DMA and CPU paths, adapting to real-time changes in DMA resource availability.

Finally, we incorporate `Fastmove` into the Linux kernel as an OS library with a limited number of simple APIs, which can be used to easily replace system functions that trigger data movements. To demonstrate its practicality, we adapt three NVM-based storage systems, `NOVA`, `ext4-DAX`, and `XFS-DAX`, to make use of `Fastmove` with minimal (2 to 4 lines of code) change. Consequently, applications running atop these systems can transparently enjoy the data movement acceleration brought by `Fastmove`. Additionally, we enable such acceleration for the cross-socket setting by deploying file systems atop the Linux device mapper with 2 lines of code change. This design enables the POSIX `read()` and `write()` APIs to freely employ `Fastmove`. To prove this, we successfully run three I/O-intensive applications, one industry-adopted database, MySQL [3], one graph engine, GraphWalker [49], and one file system and storage benchmark, Filebench [1] atop the modified file systems without any modifications to the applications.

We conduct extensive evaluations with three standard benchmarks FIO [8], fileserver [1], and TPC-C [5], and three popular random walk algorithms GraphLet, PageRank and SimRank. The results highlight that, for workloads containing substantial I/Os with moderately large sizes and beyond, considerable performance improvements are attained, regardless of local or remote NVM access. For TPC-C in MySQL, `Fastmove` increases its peak throughput by 13-116% compared to the original ones that use only the CPU, reduces the average latency by 17.7-60.8%, and saves CPU cycles used for data movement by 37.1-68.9%. Also, `Fastmove` brings 1.65-2.14× speedups of execution time for the GraphWalker algorithms, and 1.12-1.27× speedups of throughput for the Filebench fileserver workload.

In summary, `Fastmove` makes the following contributions:

- We present a comprehensive and general study to understand the characteristics of on-chip DMA in conjunction with NVM far beyond earlier studies [21], which showed DMA use just as a minor optimization in limited experimental settings [7, 25, 41].
- We propose and implement a fast memory copy engine `Fastmove` that accelerates DRAM-NVM data movement in NVM-based storage systems. Driven by the study findings, it incorporates new latency-oriented optimizations to reduce associated DMA costs and coordinates the CPU-only and DMA paths to maximize overall performance. `Fastmove`'s design principles significantly differ from earlier studies that concentrated on movement of data in a tiered memory setting [41, 54], where optimizations are simple due to the large size of memory copy requests.
- We present transparent in-kernel system support with integration of `Fastmove` into three NVM-aware DAX file systems, while extending the device mapper to enable cross-socket NVM access. This allows unmodified applications to run atop `Fastmove`.

## 2 Background and Motivation

### 2.1 NVM-based Storage Systems

NVM chips sit close to the CPU either by being placed on the memory bus and connected to CPU sockets via the processors' integrated memory controller (iMC) or by being exposed via cache coherence interconnects like Compute Express Link (CXL) [11, 18, 39]. In 2019, Intel released Optane PM, the first commercial NVM chip based on the 3D XPoint technology [15]. Beyond Optane PM, multiple companies are developing new products based on technologies other than 3D XPoint [15] such as STT-MRAM [45], FRAM [23], NanoRAM [42], and ReRAM [6].

Despite the different implementations, they are expected to offer memory interfaces with byte-addressability, data persistence, and large capacity. Therefore, there have been extensive research focusing on incorporating NVM to build scalable storage systems [9, 22, 24, 27, 34, 53] that accelerate the data access of latency-critical, data-intensive applications. These applications persist all their data on NVM, while caching the working set and metadata like indexes in DRAM. When accessing non-cached data, applications need to load them from storage, while upon modification, the dirty pages and log entries need to be flushed back to storage for data durability.

Typically, they make use of NVM-aware DAX file systems such as `NOVA` that retain the standard file system interfaces

Table 1: I/O size (KB) distribution of various workloads

| | size | TPC-C | fileserver | Graphlet/PPR/SR |
|---|---|---|---|---|
| read | [0,16) | - | 80.2% | - |
| | [16,32) | 100% | 11.5% | |
| | [32,∞) | - | 8.3% | 100% |
| write | [0,16) | 6.5% | 82.2% | - |
| | [16,32) | 82.9% | 10.2% | - |
| | [32,∞) | 10.6% | 7.6% | - |

and provide strong consistency guarantees along with various NVM-oriented performance optimizations [7, 20, 53]. Therefore, the aforementioned data copies often involve memory allocated in user space, while requiring kernel memory copy module support.

## 2.2 The Data Movement Bottleneck

DRAM-NVM data movement can be a critical bottleneck in terms of performance in data-intensive applications. To understand this, we perform a study on the I/O size distributions of various applications, from domains ranging from traditional SQL databases to graph analytic frameworks, and their impact on performance and resource usage.

As shown in Table 1, driven by the standard database TPC-C workloads with 5000 warehouses and 16KB innodb page size in MySQL, more than 93% of write I/Os in MySQL are beyond 16KB, where a significant number of these bulk writes are sitting on the critical path of writing logs for foreground update transactions. In the fileserver workload of Filebench, 8.3% and 7.6% of the reads and writes are beyond 32KB, respectively. Though the number of bulk I/Os is relatively small in fileserver, they already account for 44.1% of the overall data movement volume. Finally, GraphWalker, a single-threaded graph processing system, periodically reads from NVM into DRAM, all in 128KB chunks, which it later consumes with its in-memory processing [49].

To assess the negative impacts of data movement, we run the `msppr` workload [49] in GraphWalker atop `NOVA`, an NVM-based file system, with Optane PM. Note that `NOVA` uses Linux `memcpy` to access data on Optane and does not make use of SIMD as SIMD cannot be used within the kernel [43]. We find that over 92% of the execution time is spent on reading data from NVM under a single socket setting, while, when cross-socket data movement is involved, this number increases to over 97%. While these numbers will vary depending on the application, our observation is that for many applications, the time consumed for data movement is a clear bottleneck.

The inefficiencies of CPU-directed data movement are mainly caused by the performance gap between DRAM and NVM. In particular, with 6 interleaved Optane DIMMs within a single socket, reading a 4K page from Optane takes 952ns, 2.9× longer than that of DRAM. Similar to latency, PM shows 74.4%/35.3% lower read/write throughput than DRAM. Even worse, it takes 18 CPU cores for Optane to reach its peak load throughput while it only takes 5 for DRAM to reach a similar load throughput [55]. Finally, when accessing remote memory across sockets, both DRAM and NVM suffer negative NUMA effects due to the extra writes introduced by the default directory-based cache coherence protocol [28]. However, the performance loss of remote NVM accesses is larger because of its lower write bandwidth. Our findings are consistent with recent studies [14, 28, 55].

## 2.3 On-Chip DMA and its Challenges

Modern processors have included on-chip DMA engines since as far as one can remember. For instance, Intel's I/O Acceleration Technology (I/OAT) DMA engine [16] lies in the integrated I/O module of the CPU, which also connects to cores and iMCs through a mesh interconnect. Similarly, AMD's second-generation EPYC processors are also equipped with on-chip DMA engines [37]. With the advent of high performance storage devices, however, they have deteriorated to a mostly unused component. The observations behind the data movement overhead problem motivate us to re-think the role of the on-chip DMA in NVM-based storage systems. We advocate that it will be beneficial to use on-chip DMAs to offload data copy jobs in NVM-based storage systems, thereby improving the copy performance itself as well as saving CPU cycles that could be used for other useful work.

To explore the latency improvement potential of DMA, we evaluate the speed of moving data between DRAM and NVM achieved by Intel I/OAT, in comparison with the CPU-only counterparts. Here, we refer to the I/OAT setting as Simple-DMA as we use it as-is without optimizations, which are explored later.

We use the FIO benchmark [8] to generate single-threaded read and write requests with I/O sizes ranging from 16KB to 512KB, where the former load data from NVM to DRAM while the latter store data in the opposite direction. These requests trigger kernel memory copy functions through `NOVA` to operate the underlying NVM—Optane PM [55], and we measure the time consumed for those functions.

Figure 1a and Figure 1c show that Simple-DMA performs consistently worse than CPU-only, and delivers 29.9-134.4% higher read latency, regardless of local and remote accesses. Contrary to reads, for local writes as shown in Figure 1b, Simple-DMA delivers comparable latency as CPU-only at 64KB, with meaningful differences expanding with I/O sizes from 128KB and beyond. For instance, when writing 256KB, the latency of Simple-DMA is only 64.7% of the CPU-only latency. Compared to the single-socket results, in Figure 1d, when considering two sockets, we observe that the performance of remote writes achieved by CPU-only and Simple-DMA both worsen. However, the request size threshold where Simple-DMA catches up with CPU-only becomes smaller at 16KB, which is only 25% of that observed for local writes.

The above latency comparison suggests that there is hardly any opportunity to allow reads within NVM-based storage systems to benefit from Simple-DMA; while for large writes, opportunities seem to exist. However, whether such large

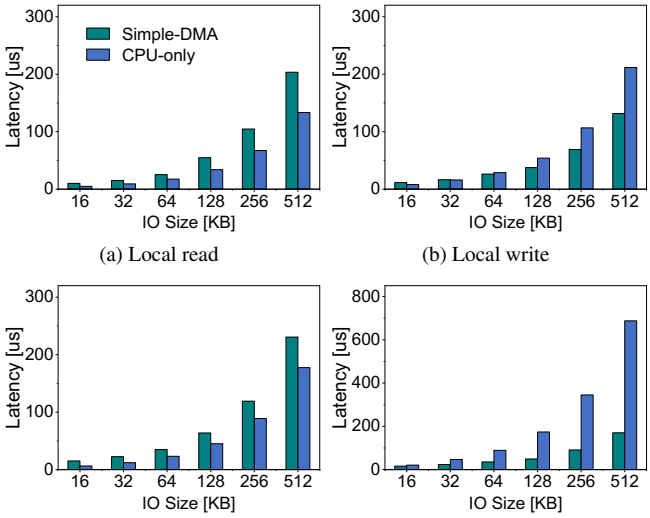



Figure 1: Simple-DMA versus CPU-only read/write latency as request size is varied with FIO workloads.

writes (not smaller than 128KB for local writes) are amply available in typical applications is questionable. For example, as shown in Table 1 in our evaluation, around 80% of the bulk writes for MySQL-TPC-C concentrate on the range of [16KB, 32KB), which is certainly below the benefit threshold of Simple-DMA. Our conclusion is that we need to explore whether there are optimization opportunities.

Moreover, we have witnessed initial adoptions [7,21,41,54] of on-chip DMA to accelerate DRAM-NVM data movement. However, these early attempts mostly focus on tiered memory systems, and cannot be directly applied to NVM-based storage systems, which is our focus, due to the following reasons.

First, our optimization goal differs from using DMA in tiered memory systems, where data movements triggered by page migration run in the background, not on the critical path of user requests. Related works primarily focus their optimization goal on deriving advanced migration policies, and use DMA as a minor optimization to free CPU cycles [41]. In contrast, for NVM-based storage systems, data copy jobs such as user reads and log flushing are part of an end-to-end execution of foreground requests, which directly affect user experience. Thus, the key performance measure is latency.

Second, the I/O patterns and workflows differ significantly between NVM-based storage systems and tiered memory systems. The page migration workloads in tiered memory systems are quite simple and always happen at 2MB huge page granularity [41, 54]. In contrast, the sizes of bulk I/Os in NVM-based storage systems are much smaller and vary considerably. It is equally important that the workflow of handling memory copies via DMA in NVM-storage systems contains considerably more steps than that of tiered memory. These differences imply that the associated overhead of DMA is not negligible in NVM-based storage systems.

In summary, the Simple-DMA performance, the demand

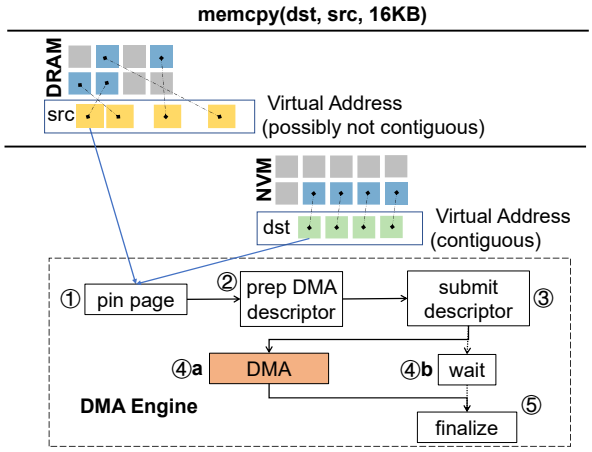



Figure 2: Workflow of memory copy using Simple-DMA.

for reducing latency and the storage-specific I/O patterns present us with unique challenges in making use of the DMA in NVM-based storage systems. In this paper, through an in-depth study of the behavior of on-chip DMA, we explore avenues of optimization opportunities. In addition, through `Fastmove`, we develop the necessary abstractions and transparent latency-sensitive optimizations so that applications may reap the benefits of the DMA without any code change.

## 3 DMA Optimization Opportunities

Here we provide a comprehensive study on DMA in conjunction with NVM to derive the optimization directions for lowering the latency of DMA-enabled memory copies and for unleashing its potentials to (partially) alleviate the above DRAM-NVM data stall problem.

### 3.1 DMA-enabled Data Moving Workflow

To begin our study, we first illustrate in Figure 2 the workflow of handling memory copy requests issued by applications via DMA, which implements exactly the same logic as the Linux `memcpy`. Take a 16KB I/O as an example. The virtual addresses of data residing in DRAM for NVM-based storage systems are possibly not contiguous, which leads to this single memory copy operation at the application side being divided up into four DMA subtasks. Each subtask corresponds to a 4KB page and will go through the following steps. ① pins the target DRAM pages as we need to prevent those pages from being swapped out or modified during DMA execution. An alternative way to do so is to allocate a DMA buffer, but at the cost of imposing extra memory copies or giving up transparent support to applications. ② prepares the DMA descriptor, the required metadata for I/OAT, which is then submitted to the hardware at step ③. Meanwhile, the submitter waits (④b) until the completion of ④a and reaches the final step ⑤ to finalize the corresponding DMA subtask execution, e.g., unpinning the page and notifying the application. Note that all steps except ④a are managed by a CPU thread, often the I/O thread of the application.

Table 2: Breakdown time costs of local read and write requests that use Simple-DMA

| | size (KB) | cost (%) | | | | #subtasks |
|---|---|---|---|---|---|---|
| | | pin | submit | I/OAT | other | |
| read | 16 | 4.7 | 12.7 | 80.4 | 2.2 | 8 |
| | 32 | 4.9 | 14.1 | 78.7 | 2.3 | 16 |
| | 64 | 5.1 | 14.8 | 77.9 | 2.3 | 32 |
| write | 16 | 6.2 | 15.7 | 75.3 | 2.9 | 8 |
| | 32 | 7.1 | 19.8 | 69.8 | 3.3 | 16 |
| | 64 | 7.9 | 23.3 | 65.3 | 3.5 | 32 |

## 3.2 I/OAT and Optane PM Demonstration

To make the study concrete, in this section, we focus on the combination of Optane PM and Intel's I/OAT DMA.

### 3.2.1 Associated Time Costs

First, we investigate the latency breakdown results of Simple-DMA, which are summarized in Table 2, with the same setup as Figure 1a and Figure 1b. "pin", "submit", and "I/OAT" correspond to steps ①, ②-③, and ④a of Figure 2, respectively, while "others" denotes the remaining overhead.

The execution on the I/OAT hardware is the longest step of DMA-enabled memory copy requests across reads and writes. However, its ratio decreases from 80.4% to 77.9%, and 75.3% to 65.3% for reads and writes, respectively, when I/Os expand from 16KB to 64KB. In contrast, the associated overhead, excluding I/OAT, is also non-negligible and grows proportionally with request size, reaching to 34.7% for local 64KB writes. This is mainly because bulk I/Os within NVM-based storage systems trigger a series of I/OAT subtasks at 4KB granularity, as introduced in Section 3.1.

This growing overhead can be further doubled when the source and destination addresses of the corresponding I/O request are not aligned. Figure 3 illustrates such an example. The *src* of page#1 is not aligned with *dst* of page#a. As the DMA does not support cross-page copy when it cannot tell if the physical address is contiguous between pages, we have to split page#1 into two separate portions, namely ① and ②, where the former fits in the empty space of page#a, while the latter will have to fit on the lower part of page#b. Each of these portions will trigger a separate I/OAT subtask. Moreover, the remaining two pages #2 and #3 will go through the same effort. As the FIO workloads exhibit unaligned memory addresses, as shown in Table 2, bulk I/Os consist of 8-32 DMA subtasks and pay the associated time cost one more time. As this shows, in the case of transferring unaligned memory addresses, the overhead involved can turn out to be even more significant.

Trimming down the associated costs seems promising for improving the latency of writes. For instance, one can imagine that reducing them by 30.7% for 16KB writes will allow the DMA latency turning point to be reduced from 64KB to 16KB, enabling more applications, like MySQL, to gain performance benefits. However, this is not so with reads, since even completely eliminating these overheads still results in the DMA performing 11.1%-39.3% slower than CPU-only. Thus, other means to overcome this challenge must be conceived.

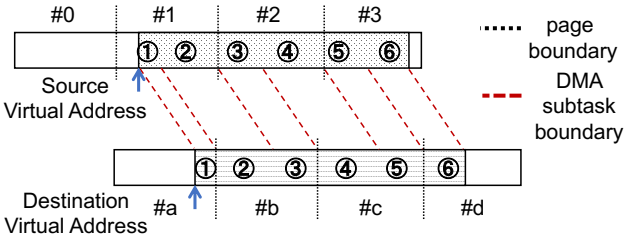


Figure 3: Composition of DMA subtasks for a single application data movement job with unaligned source and destination addresses. Three source pages (#1-#3) are involved but six subtasks are generated (①-⑥).

In addition, using Transparent Huge Page (THP) in the kernel makes the addresses, with high probability, to be contiguous. For contiguous copies, the cost of I/OAT still dominates, but with the submission and unalignment cost significantly diminished, compared to the above non-contiguous ones. This is because under such setting, memory copy requests will no longer be divided into multiple DMA subtasks.

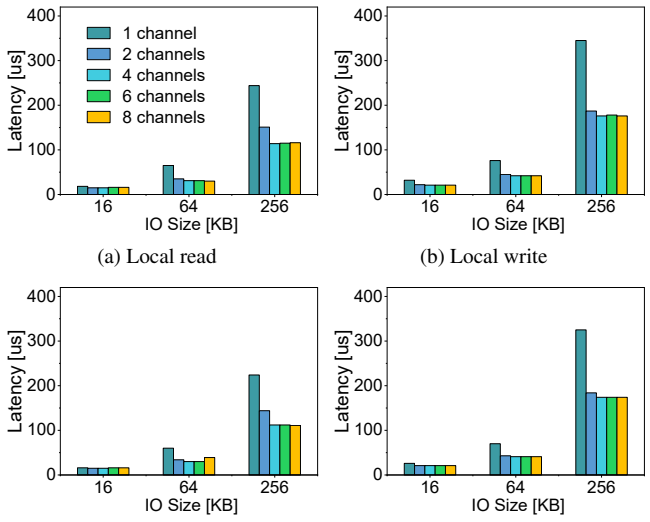
### 3.2.2 Intra-Request Parallel Copy

Each DMA device consists of $M$ multiple channels that can process DMA subtasks in parallel. Therefore, we explore parallelizing hardware copy of a single request, where we split the request into $N$ chunks ($N \leq M$) and thus, $N$ DMA subtasks, each chunk making use of one channel. Here, we derive two different parallel execution modes, namely, para-A and para-B, where para-A uses a single submitter for channel submission, while para-B spawns $N$ submitters, each of which manages its own channel independently.

For 64KB reads and writes, compared to Simple-DMA, para-A indeed reduces the I/OAT copy time, but the reduction is not proportional to the number of parallel chunks. In addition, we observe a significant increase in the submission overhead, which eventually offsets the benefits of intra-request, multi-channel parallel copy. In the end, para-A does not improve much on the end-to-end latency of Simple-DMA for reads, while even leading to performance loss for writes.

Para-B fares worse than para-A, worsening latency for both reads and writes. Our analysis shows that para-B sharply increases the hardware copy time by up to 68.7%. This is because of the heavy contention on DMA bandwidth driven by the parallel subtasks. This case differs from para-A, as the single submitter setting in para-A enables pipeline parallelism, which does not heavily stress the DMA. In addition, para-B introduces heavy CPU usage due to the multiple submitters.

Finally, as we cannot parallelize intra-request copies within DMA, we also explore the possibilities of balancing these copy subtasks between the CPU and DMA. Unfortunately, this is not applicable for writes, as using the DMA can easily saturate NVM's bandwidth. We find that the bandwidth competition can lead to amplified interference between the two tasks, resulting in 14.6% higher latency compared to the sole execution of using the DMA. In contrast, we find this



(a) Local read (b) Local write

(c) Remote read (d) Remote write

Figure 4: I/OAT latency when 4 FIO threads doing read/write workloads as number of channels and as request size is varied.

solution works well for reads as the DMA cannot consume all of the NVM bandwidth, and thus, the joint use of the CPU and DMA leads to better bandwidth consumption. We take this last approach as part of our optimization.

#### 3.2.3 Impacts of Inter-Request Parallelism

Finally, we evaluate the impact of inter-request parallelism as in reality, application threads may concurrently execute data movement requests and make use of the DMA. First, we investigate whether using more DMA channels influences the performance of DMA operators. To this end, we use four concurrent threads to submit DMA requests to its multiple DMA channels on our two-socket NUMA machine. Here, we exercise up to 8 channels per DMA device/NUMA node. Figure 4 shows the latencies of DMA operators with varied I/O sizes. With the increasing number of channels, irrespective of local/remote reads/writes, the DMA operators become faster. For instance, compared to the 1 channel setting, adding one more channel leads to 38.1%-53.3% latency reduction for the 256KB memory copy operators. Trends are similar with more concurrent threads and cross-socket NVM accesses.

Second, we explore the changes in read/write effective bandwidth with the increase in the number of concurrent threads submitting DMA requests with bulk I/Os. We find that Simple-DMA observes an increase in read/write effective bandwidth for up to four threads, but beyond this, it starts to decline sharply. (Results not shown due to space limit.) The key limiting factor here is not drive scalability but, instead, the I/OAT DMA bandwidth. This suggests that a limit on concurrent DMA access should be set to prevent the DMA resource from being over-used.

### 3.3 Study Generalization

While the performance study above takes into consideration the performance characteristics of the underlying hardware, it also lays out the general study flow and key factors to be considered independent of particular NVM and DMA devices. With the advent of new hardware, the general study always needs to answer the following two questions:

First, *how can the DMA be best configured so that using it can be faster than CPU-only even for small I/O requests?* This part requires understanding the DMA subtask associated cost, the DMA parallel execution, and the effects of concurrency that drive the latency-oriented optimizations. Furthermore, it also requires exploring the effects of balancing loads among DMA channels and even between DMA and CPU.

Second, *how do we choose among the different copy paths?* We decide the best-effort path with the minimal time cost among three choices, namely, CPU, DMA, and DMA-CPU cooperation. Furthermore, we have to check if there are available DMA resources, i.e., the current DMA bandwidth usage, monitored during DMA execution, is below the profiled maximum bandwidths of DMA and NVM, respectively.

In summary, our general study framework will offer useful guidelines for accelerating data movement in storage systems that combine future DMA implementations and near-DRAM storage devices such as the upcoming CXL devices.

## 4 Overview of `Fastmove`

Driven by the study in Section 3, we aim to let data-intensive applications transparently make the best use of DMA to alleviate the NVM data stall problem presented in Section 2.2. Done properly, this should lead to better performance and alleviate CPU involvement required for memory copies between DRAM and NVM. First, we need to improve the latency of DMA-enabled data movement by taking into consideration the access constraints of DMA such as extra overhead, resource allocation, and interference within DMA or with CPU. Second, to complement DMA's limitations, we need to judiciously determine when and how much to resort to the normal CPU data path. Finally, while DMA is supported by Linux kernel functions, applications should not be burdened by high development and optimization overhead to exploit the DMA. Thus, a clean abstraction that requires minimal changes to applications is imperative.

### 4.1 `Fastmove`'s Architecture

Figure 5 shows the overall design of `Fastmove`, our efficient data movement engine. It sits below DAX file systems such as `NOVA`, `ext4-DAX`, and `XFS-DAX`, which are compatible with POSIX APIs and designed to use recent PM, as well as the Linux device mapper module, which allows file systems to use PMs across sockets. With this design, applications that run atop a POSIX file system should seamlessly be able to use our engine. `Fastmove` consists of three major system components, namely, `Scheduler`, DMA module, and CPU module. We retain the original design of the CPU module, where we let the corresponding I/O request execute the `load` and `store`

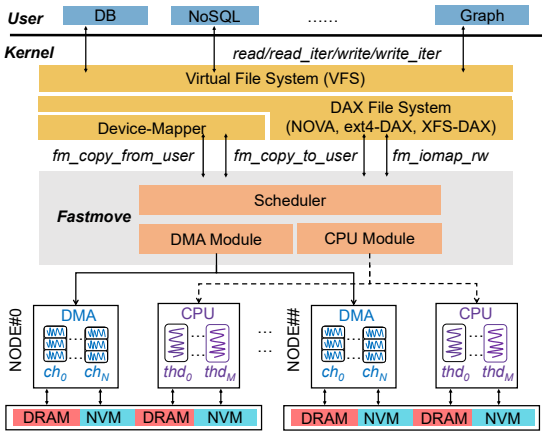



Figure 5: The overall architecture of `Fastmove`, which manages both DMA and CPU resources. Each NUMA node has a DMA device (dashed line box), which has multiple channels.

instructions as usual. However, we introduce a new DMA module that manages DMA resource allocation and memory copy offloading, with various optimizations to alleviate DMA costs and improve DMA resource usage. (Details will be discussed in Section 5.1.)

As the core logic, `Scheduler` is responsible for making decisions on selecting either DMA or CPU to execute requests. Its detailed design will be discussed in Section 5.3. This decision-making procedure should be fast so as not to incur overhead on the end-to-end request latency. It should also be smart so as to prioritize the use of DMA to fully make use of its strengths, while resorting to the CPU-only path as needed to compensate for the limitations of DMA for overall enhanced performance (Section 5.2).

## 4.2 API Abstraction

To exploit DMA transparently at the application level, we introduce three APIs that are simple extensions to existing APIs used by DAX file systems. The key observation here is that DAX file systems universally make use of a limited number of APIs for data movement, namely, `copy_from_user`, `copy_to_user`, and `dax_iomap_rw`. The first two are called by the `read` and `write` file system functions, while the last API is used by the `read_iter` and `write_iter` file system functions to perform memory copies in batches. These APIs are replaced by the APIs that we describe below.

As shown in Table 3, the three APIs that we introduce are `fm_copy_from_user`, `fm_copy_to_user`, and `fm_iomap_rw`. The first two new APIs have four arguments, `dst`, `src`, `len`, and `bdev`. `dst` and `src` specifies the destination and source of the copy (from PM to DRAM or vice versa), while `len` refers to the number of bytes to copy. The last argument `bdev` is the PM block device descriptor that includes rich information of the PM device such as the NUMA node id of the target PM. The last API `fm_iomap_rw` has three parameters, where `iocb` specifies the operational semantics such as read or write, `iov_iter` encodes parameters such as

Table 3: `Fastmove` APIs

```
fm_copy_from_user(dst, src, len, bdev);
fm_copy_to_user(dst, src, len, bdev);
fm_iomap_rw(iocb, iov_iter, iomap_ops);
```

source and destination address vectors, and `iomap_ops` that is passed by file systems for I/O address mapping.

Finally, we only need to replace the old APIs with the new ones at the file system level. Thus, upper layer applications can take advantage of `Fastmove` without any code change. (Details are discussed in Section 5.4.)

# 5 Design and Implementation

## 5.1 High-Performance DMA Module

Under `Fastmove`, we offer a dedicated wrapper module to easily use the low-level primitives that DMA offers. This wrapper executes the I/O requests admitted by `Scheduler`. Here, we encapsulate the DMA requests by inheriting the values of parameters from the `Fastmove` APIs and the DMA channel assignment from `Scheduler`. Then, the wrapper executes DMA requests by going through all the steps in Figure 2 with the following major techniques and optimizations.

**Batched DRAM page pinning.** Memory addresses passed from user space are all virtual and need to be translated into physical ones that the DMA can consume. Furthermore, to satisfy DMA requirements, the virtual-to-physical address mapping must remain valid and unchanged during the execution of the corresponding DMA copy. This can be done by calling the `pin_user_page` and the `dma_map_page` kernel functions. However, pinning user pages one by one incurs high overhead for bulk I/Os, which span across multiple pages. To lighten this overhead, we leverage the `pin_user_pages` function available in the recent Linux kernel (version 5.9) that pins all the pages belonging to a single I/O. Similarly, we apply the same optimization for `unpin_user_page` via the new `unpin_user_pages` function.

**src/dst page pairing.** A bulk I/O will be mapped to a list of DMA subtasks at 4KB page granularity, each of which requires to pair the addresses of the source and destination pages for preparing the DMA descriptor. If the two addresses are not aligned, to ensure the correctness of DMA execution, which assumes that copies take place within page boundary, we have to carefully match the capacity of *dst* pages and the content size of *src* pages so that cross-page copies can be avoided. However, this leads to doubling the number of DMA subtasks, as described in Section 3.2.1.

Here, we make a key observation that NVM is managed contiguously in the kernel, and thus the cross-page copies can be tolerated. We exploit this finding as when preparing the DMA descriptor, we specify the length of the corresponding subtask in a page aligned manner on the DRAM side. For instance, take the situation in Figure 6 assuming that the source and destination are DRAM and NVM, respectively. We take the first portion of the source (① of page#1), which

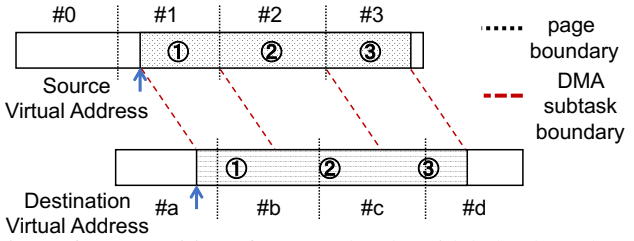


Figure 6: Composition of DMA subtasks with halved numbers for a data movement job, enabled by the contigious NVM address management, in comparison to Figure 3.

will always be smaller or equal to a page but aligned on the right end, as the size of the first DMA subtask. Thereafter, the size of the subsequent DMA subtasks will always be a page and aligned (② of page#2) except possibly for the last portion (③ of page#3), which will be page aligned on the left end. This enables us to reduce the number of DMA subtasks by half, in comparison to Figure 3.

**Metadata buffer pre-allocation.** DRAM space must be allocated with varying sizes to store the DMA request metadata, i.e., descriptors. The `scatterlist` structure is used to store the list of descriptors of DMA subtasks belonging to a single bulk I/O, where each item is typically 32 bytes. To accelerate memory allocation, we pre-allocate a fixed-size buffer to store this information prior to the execution of DMA copy. We set the buffer size to 4KB, which can accommodate DMA request metadata for 128 user pages (in total 512KB) at once.

**Batch submission.** Finally, to amortize the DMA subtask submission, considering that leveraging multiple channels performs no better than using a single channel (Section 3.2.2), we submit `scatterlist` in a batch to a single DMA channel assigned by `Scheduler`. This batched submission reduces the locking overhead for coordinating the concurrent accesses of the task queue associated with the DMA channel [17].

## 5.2 DMA-CPU Cooperated Bulk Reads

With Simple-DMA, the application thread (CPU) submits requests to the DMA, which solely moves the data (see top part of Figure 7). However, as shown in Section 3.2.3, bulk reads could be made faster through DMA and CPU cooperation. Motivated by this, we design an optimized bulk read within `Fastmove` that is depicted by the lower part in Figure 7. Here, the application thread first splits the bulk read into two chunks, and then submits one chunk (#1) via the normal DMA path with optimizations mentioned in Section 5.1, followed by the other chunk (#2) being copied by the CPU. Upon completion of chunk#2, the corresponding CPU thread polls the status of the DMA. Finally, the execution of the target read completes when both the DMA and CPU finish their assigned chunks. This design not only improves the NVM read bandwidth but also hides the copy latency due to the CPU.

While the optimized bulk read is a natural sharing of load, the challenge we face here is how to decide the loads that will go through the CPU and DMA. Chosen inappropriately,

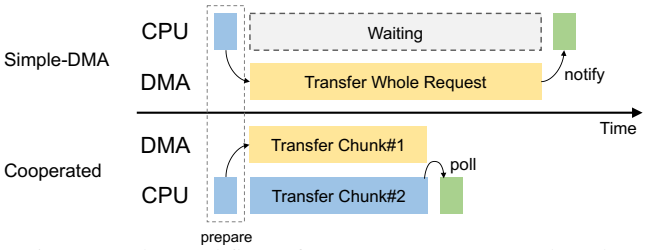


Figure 7: The workflow of DMA-CPU cooperated reads.

the gap between the execution time of CPU and DMA could lead to either waste of CPU cycles for polling the DMA status or lower DMA utilization. To balance their execution time, we set the chunk #1 and #2 size ratio to the ratio of the average single-threaded bandwidth on the CPU and DMA paths. which are monitored by our `Scheduler`.

## 5.3 Controlling and Scheduling

We design a light-weight `Scheduler` that outputs the proper memory copy path assignment plan for each I/O request going through the above `Fastmove`'s APIs, distributes loads of bulk reads between CPU and DMA, and properly allocates DMA resources for offloaded tasks.

**Initial configuration.** Decision-making by `Scheduler` is driven by the four pre-chosen I/O size thresholds for local/remote NVM reads/writes, beyond which DMA path should be involved for better performance, and the concurrency sweet spot $M$ per DMA device, which corresponds to the maximal number of concurrent threads leading DMA to reach the peak bandwidth. In addition, `Scheduler` also monitors the following four variables: (1) $C_i$, which is used to keep track of the number of on-the-fly requests submitted to device $i$ and that works as an indicator of the workload intensity level of that device; (2) $S_i$, which points to the next available DMA channel on the DMA device $i$; and (3) $B_C$ and $B_D$, that record the bandwidth dynamically consumed by the CPU and DMA, respectively.

**Scheduling.** `Scheduler` first inspects every I/O request to figure out the following parameters: the NUMA node id of the target NVM ($N_P$), the request type $RW$, the NUMA information $LR$, and I/O length $L$. $RW$ and $LR$ are both boolean values indicating read/write and local/remote, respectively. Then, the path scheduling logic is straightforward as follows. `Scheduler` compares the request length $L$ to the DMA threshold, pointed by the pair of $RW$ and $LR$, to identify bulk I/Os. For bulk I/Os, `Scheduler` chooses the DMA as long as the DMA device on node $N_P$ is under its concurrent limit, i.e., $C_{N_P} < M$. If so, `Scheduler` chooses the next DMA channel associated with $N_P$'s DMA in a round-robin fashion (based on $S_{N_P}$) and updates the required resource variables, i.e., $C_{N_P} = C_{N_P} + 1$ and $S_{N_P} = (S_{N_P} + 1)\ mod\ G$. Otherwise, we fall back to the CPU-only data path. Additionally, we use $B_C$ and $B_D$ to derive the split ratio of bulk reads between the CPU and DMA by following the logic presented in Section 5.2.

**Performance consideration.** To minimize the overhead that

may incur due to request processing, we make the following two design choices. First, instead of implementing the Fastmove logic as a centralized component for coordination, we provide the logic as a function, which runs at the memcpy caller side. This precludes inter-thread communication between I/O threads and Scheduler helping enhance performance. Second, coordination of concurrent access to globally shared variables like $S_N$ and $C_N$ adopt lightweight mechanisms such as atomic counters to further reduce overhead.

## 5.4 Implementation Details

We implement Fastmove[1] under the DMA framework [32] in Linux kernel 5.9 with 2417 lines of C code for its core logic.

**Integration with NVM-based storage systems.** We integrate Fastmove into three widely-adopted, DAX file systems, namely, NOVA [53], ext4-DAX [30], and XFS-DAX [33], where NOVA is tailored for hybrid DRAM-NVM settings, while the other two systems are more general and compatible with NVM. Fastmove's transparent design leads to minimal changes to the above systems. Specifically, we introduce only 2 lines of code changes to both ext4-DAX and XFS-DAX, which simply replace the memory copy functions in read_iter() and write_iter() system calls with the APIs in Table 3. NOVA requires 2 additional changes to its read() and write() functions.

Though Fastmove enables NUMA NVM access by design, DAX file systems cannot naturally use NVM devices sitting across NUMA sockets. We address this problem by leveraging the Linux native device mapper [31], as shown in Figure 5. For the device mapper, similarly, only 2 lines in dm_copy_from_iter and dm_copy_to_iter functions need to be replaced. Note, however, that the current version of NOVA does not support the use of the device mapper. Therefore, we extend NOVA to work with the device mapper and its new code base can be found in Fastmove [1]. With these minimal changes, Fastmove is able to transparently benefit many applications that run atop these three file systems.

**Correctness guarantee.** The use of DMA in Fastmove will not introduce any data inconsistencies compared to CPU-only data accesses. First, while not mentioned in any public documentation from the hardware vendor, Kalia et al. [21] experimentally show that I/OAT preserves ordering during execution. Second, Fastmove always monitors the execution status of parallel DMA subtasks and knows which set of pages failed to be copied even though these pages may not be consecutive. This slightly relaxed memcpy semantic is enough since (1) most applications including filesystems and databases have their own well-designed fault handling mechanism, which can leverage Fastmove 's fault reports to recover state correctly, and (2) in kernel, there are many strict checks to avoid copy failures, such as permission validation prior to copy execution. Thus, failures will be rare.

[1]Publicly available at https://github.com/fastmove-open/fastmove

# 6 Evaluation

## 6.1 Experimental Setup

We deploy our experiments on a physical server with two 20-core Xeon Gold 6248 processors and 192GB DRAM. This machine has two NUMA nodes, each connected with six Intel Optane PM chips (128GB each and 1.5TB in total). We evaluate Fastmove with both the Optane PM device and emulated NVM to demonstrate the generality of Fastmove. With Optane PM, we configure it to be interleaved within each NUMA node and under the App Direct mode, and use the Linux device mapper under its striped mode to enable cross-socket NVM accesses. For the NVM emulated experiments, we use 64GB DRAM to emulate an advanced NVM device with DRAM-like latency and bandwidth, which is significantly better than Optane PM, using a Linux built-in emulator [35]. Note that our evaluation primarily focuses on Optane PM, while the emulated NVM performance results are only presented in Section 6.4.1.

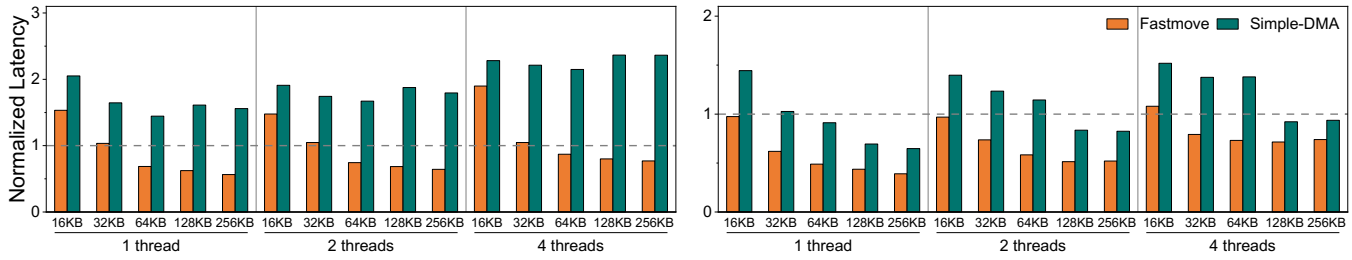
**Baseline and configurations.** We exercise NOVA, ext4-DAX, and XFS-DAX enhanced by Fastmove. Our natural baselines are these file systems with their memory copy operations going through the conventional CPU path, denoted by "CPU-only". We use default configurations for both baselines.

**Case study applications and workloads.** We take three data-intensive applications MySQL, GraphWalker and Filebench, with no code changes, to transparently use Fastmove by simply running them atop the three slightly modified DAX file systems. To evaluate Fastmove's benefits, we run experiments with the FIO microbenchmark [8] and a synthetic workload generated based on FIO, application workloads like the widely-adopted standard database workload TPC-C [5] and the file access workload fileserver [1], and three popular graph processing tasks, namely, Graphlet Concentration, Personalized PageRank and SimRank. The detailed configurations are presented in Section 6.3.

## 6.2 Microbenchmark Results

### 6.2.1 Latency Threshold Choices

To help figure out the read/write thresholds with different concurrency levels required to drive the memory copy path selection in Fastmove, we run the FIO workloads to evaluate both the original and modified NOVA file systems. Here, we generate read and write workloads with different I/O sizes ranging from 16KB to 64KB, which are issued by 1 to 4 concurrent threads. We test both local and remote (cross-socket) NVM accesses. In Figure 8, we show the normalized average latency of Fastmove against the CPU-only baseline. (We omit the results for remote NVM access as they show similar trends with the local accesses.) In addition, we also include the results of "Simple-DMA", the baseline with DMA enabled but not highly optimized, to demonstrate the validity and effectiveness of Fastmove's optimizations and our Fastmove. Note that, these results look exactly the same across three




(a) Local read (b) Local write

Figure 8: The latency comparison between `Fastmove` and Simple-DMA, with 1,2,4-threaded FIO workloads, normalized to the latencies of CPU-only memory copying. (Remote read/write results omitted due to similar trends and space limit.)

file systems, thus we omit the evaluation of `ext4-DAX` and `XFS-DAX` for this part.

As shown in Figures 8a, across all exercised I/O sizes, CPU-only delivers constantly lower local read latency than Simple-DMA. In contrast, `Fastmove` visibly improves the performance of Simple-DMA and introduce 1.20-3.07× speedups for various I/O sizes, leading read requests with relatively small sizes to benefit from DMA. Compared to CPU-only, the turning points of `Fastmove` are 32KB across the 1, 2 and 4 threaded workloads, respectively. Including and beyond these turning points, `Fastmove` starts to observe a visible reduction in average request latency. For instance, `Fastmove` reduces the local read latency of CPU-only accesses by 13.0-25.6% for 64KB. While not shown, for remote read latency requests with I/O size starting with 32KB can benefit from `Fastmove` while 64KB for Simple-DMA.

For writes, we observe larger improvements than reads. Figure 8b shows that for local writes, Simple-DMA runs faster than CPU-only at 64KB, 128KB, and 128KB for the three concurrency settings, respectively. `Fastmove` dramatically improves Simple-DMA's latency, and drops the turning points to 16KB, 16KB, and 32KB. With 2 threads, `Fastmove` achieves 36.9%-49.0% and 26.3%-48.6% reduction on average latency for I/Os at 32KB and beyond, compared to Simple-DMA and CPU-only, respectively. The benefits of the two DMA variants further expand for remote writes (again, not shown). First, they perform better than CPU-only for even 16KB. Second, the latency gap between the DMA usage and CPU-only becomes visibly larger, e.g., for 256KB cross-socket I/O requests, Simple-DMA and `Fastmove` reduce latency by 75.3% and 86.1%, respectively, compared to CPU-only. Third, `Fastmove` significantly outperforms Simple-DMA by 40.7-48.1%, 65.9-73.7%, and 86.7-96.2% for 16KB, 32KB, and 64KB, respectively.

Finally, Table 4 illustrates the impact of the batched submission optimization on tail latency. We find that batching within `Fastmove` does not prolong, but rather, improves tail latency. For instance, with the same setting of 2-thread experiments in Figure 8, the P99 latency numbers in Table 4, indicating a 8.0-38.5% reduction, compared to the non-batching baseline. This is because `Fastmove` is not batching DMA subtasks across I/O requests from upper applications but is batching submissions of DMA subtasks that belong to a single request.

Table 4: P99 latency (us) comparison of local read/write with batching enabled or disabled in `Fastmove`, corresponding to the same setting of 2-thread experiments in Figure 8.

| size | read | | write | |
|---|---|---|---|---|
| (KB) | batching | non-batching | batching | non-batching |
| 16 | 8 | 13 | 10 | 11 |
| 32 | 9 | 11 | 14 | 19 |
| 64 | 16 | 21 | 23 | 30 |
| 128 | 27 | 37 | 46 | 55 |
| 256 | 49 | 72 | 80 | 87 |

### 6.2.2 Breakdown Analysis

We use two synthetic FIO workloads to investigate the performance improvements introduced by each individual optimization within `Fastmove`. The bulk dominating workload contains I/Os with an average size of 256KB, while the mixed one has a mixture of bulk and small I/Os, ranging between 8KB and 256KB. For the two workloads, we use 6 concurrent threads to issue local read or write requests to the underlying `NOVA` file system.

Figure 9 reports the normalized throughput numbers, which indicate that different workloads see different optimization sweet points. The direct usage of DMA with loads evenly distributed among channels leads to a 46.6% and 25.7% throughput drop for the bulk and mixed workloads, respectively, compared to CPU-only. This is because small I/Os do not benefit, yet still go through the DMA, and the associated DMA overheads have not yet been ameliorated. As we start to avoid overloading the DMA resources by adding the concurrent limit optimization (here, set to 4), `Fastmove`'s performance improves by 23.0% and 16.5% for the two workloads. The batching optimization makes `Fastmove` begin to outperform CPU-only, with a throughput increase of 55.8% and 17.9%. The latency threshold filtering further improves `Fastmove`'s performance by 0.3% and 9.5%, where the mixed workload observes larger improvements as this optimization avoids its small I/Os from paying the latency penalty of going through the DMA. Finally, the bulk read split design choice brings another 12.3% and 3.1% improvement. In the end, adding all these optimizations together brings a 1.15× improvement in throughput for the two workloads, compared to CPU-only.

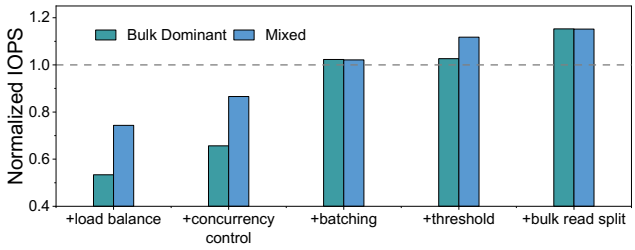



Figure 9: Breakdown analysis of `Fastmove` with synthetic FIO workloads when gradually enabling optimizations. Throughput is normalized to the CPU-only baseline.

## 6.3 Overall Performance

Next, we evaluate the positive impact of using `Fastmove` on the performance of real-world applications that introduce more complex characteristics than microbenchmarks such as non-uniformed I/O size distribution, computation-related cost, foreground and background processing division, etc.

### 6.3.1 Application Configurations

**MySQL.** We install MySQL version 5.7.33 with the default 16KB `innodb_page_size`. `innodb_buffer_pool_size` is set to half of the DRAM space, the recommended setting. We run the TPC-C workload with a read and write ratio of 1.78:1. For each run, we populate a 466GB database with 5000 Warehouses during the initialization phase and use 14 connections during the evaluation phase.

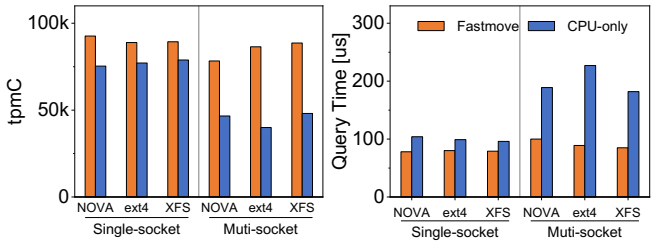
**GraphWalker.** GraphWalker [49] supports fast random walks on large graphs with a single machine. We exercise three common random walk algorithms, namely, Graphlet Concentration (Graphlet), Personalized PageRank (PPR) and SimRank (SR). We also follow GraphWalker to generate a Kron30 dataset using the Graph500 Kronecker [2], which consists of 1 billion vertices and 32 billion edges that take 638GB and 136GB of persistent media space to store its original text data and the compressed CSR data, respectively. We use the GraphWalker default configurations.

**Fileserver.** We exercise the predefined workload, fileserver, within the Filebench framework [1]. It uses 8 concurrent threads to issue I/Os with variable sizes presented in Table 1.

**Enabling/disabling THP.** We test MySQL and Fileserver without using transparent huge pages (THP), resulting in non-contiguous memory copies. This is recommended by the MySQL official site as THP introduces negative performance impacts on random memory accesses with small I/O sizes. Contrary, we enable THP for GraphWalker with contiguous copies, as its workloads are read-dominating and bulk-sized.

### 6.3.2 MySQL Enhancement

**Single-socket results.** First, we consider the performance within a single socket, where application threads and PM are located under socket 0. Figure 10a shows the throughput comparison (officially measured as tpmC by TPC-C) between CPU-only and `Fastmove` execution of MySQL. Across all settings, `Fastmove` consistently delivers better performance than




(a) Peak throughput (b) Query time

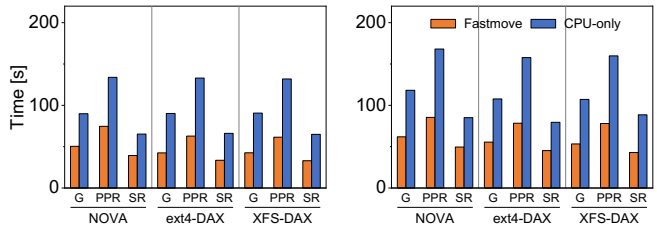
Figure 10: Throughput (measured as tpmC) and query time achieved by running TPC-C against MySQL.

CPU-only, and the improvements associated with different underlying file systems look similar. For instance, `Fastmove` introduces 1.23×, 1.15×, and 1.13× speedups of peak throughput over the CPU-only baseline across `NOVA`, `ext4-DAX`, and `XFS-DAX`, respectively. Figure 10b reports the corresponding average query time results. Consistent with the throughput results, `Fastmove` reduces the average latency of CPU-only by 17.7-25.0%.

To understand the source of improvements, we profile the I/O distribution of the TPC-C workload. As shown in Table 1, almost all of its read requests are smaller than 32KB. As this is below the 32KB threshold, the vast majority of read requests go through the ordinary CPU-only path in `Fastmove`. As a consequence, the performance improvements here are driven by the 90.9% of bulk writes beyond 16KB, which correspond to the logging activities handled by the 4 background flush threads. To conclude, `Fastmove` indeed choose proper memory copy paths for I/O with varied sizes, and I/OAT DMA does alleviate the NVM accessing data stalls.

**Multi-socket results.** Next, we explore the performance implications under two sockets, where we replicate the above experiments by evenly distributing application threads to two CPUs and spreading the data on all 12 PMs via Linux device mapper under its stripped mode.

Figure 10a shows the absolute throughput numbers achieved by CPU-only with two sockets decrease by 38.1-48.1%, compared to the single-socket counterparts. This is because performance degrades for cross-socket memory copy operations as depicted in Figure 8. In contrast, `Fastmove` observes lighter negative impact of cross-socket NVM access with only 0.8-15.5% drop in peak throughput. `Fastmove` significantly outperforms the CPU-only baseline, introducing 1.68-2.16× tmpC improvements. Additionally, in Figure 10b, `Fastmove` brings a significant latency reduction of 47.1-60.8%. Contrary to the single-socket results, we see that `Fastmove`'s improvements over CPU-only expand. This is because the threshold for remote reads drops to 16KB, which allows for cross-socket NVM reads to take advantage of the DMA if DMA usage is not full, and also the DMA benefits for remote reads and writes are larger than those for local ones.




(a) Single-socket execution time (b) Multi-socket execution time

Figure 11: Execution times running Graphlet (G), PPR and SR over GraphWalker with NOVA/ext4-DAX/XFS-DAX.

#### 6.3.3 GraphWalker Enhancement

**Single-socket results.** Figure 11a shows the execution times of three graph analytic tasks over GraphWalker. The performance of the CPU-only baseline looks similar across different file systems, and so does our Fastmove. However, we observe that Fastmove significantly reduces the execution time over CPU-only, despite the fact that the graph analytic jobs are read-only workloads towards the underlying data systems. More specifically, Fastmove introduces 1.78-2.13×, 1.79-2.14×, and 1.65-1.97× speedups for Graphlet, PPR, and SR, respectively. The significant improvements come from the dominating bulk read I/Os as shown in Table 1.

**Multi-socket results.** Consistent with the above TPC-C results, the improvements of Fastmove for the graph analytic workloads become larger compared to the single-socket counterparts. Figure 11b depicts that Fastmove brings 1.91-2.01×, 1.97-2.05×, and 1.71-2.06× execution time speedups for Graphlet, PPR and SR running in GraphWalker, respectively, across three different NVM-based file systems.

#### 6.3.4 Fileserver Enhancement

The performance trends of the fileserver workload within Filebench atop NOVA look similar to those of TPC-C and graph algorithms above. Due to the space limit, we omit the figures. To summarize, Fastmove introduces 1.12× and 1.27× speedups in peak throughput, measured by IOPS, for the single-socket and multi-socket settings, respectively.

#### 6.3.5 CPU Consumption Improvement

Finally, we explore another possible benefit of using Fastmove, which is the CPU consumption improvement. Here, we measure the CPU cycles spent in moving data between DRAM-NVM and processing the application logic. For MySQL TPC-C workload, Fastmove reduces its data movement CPU usage from 62% to 39% and from 90% to 28% for single-socket and multi-socket settings, respectively. We also observe a significant increase in its utime. This is because the saved CPU cycles from data movement are used to perform useful work, leading to improved throughput numbers (presented in Section 6.3.2). Unlike this, for GraphWalker, Fastmove's CPU usage improvement seems little. For instance, Fastmove reduces its CPU usage for data movement by up to 5%. This is because workloads with GraphWalker benefits largely by the DMA-CPU cooperated bulk read optimization, which requires CPU involvement.

### 6.4 Other Factors

#### 6.4.1 Emulated NVM Performance

We deploy NOVA on emulated NVM, replicate the experiments for Figure 8, and report the latency comparison results between CPU-only, Simple-DMA, and Fastmove in Figure 12. Fastmove outperforms CPU-only for local reads and writes with I/O sizes of 16KB and beyond. The benefits observed are larger than those corresponding to experiments with Optane PM (Figure 8). This is because emulated NVM is of DRAM-like read and write performance. Considering the association cost in Section 3.2.1, the dominating DMA copy execution step becomes faster, leading to visible end-to-end read/write latency improvements. Also, this implies that the time cost of NVM device access plays a key role in assigning DMA resources, i.e., the performance turning point based on I/O size decreases when NVM device performance improves, and vice versa. Furthermore, we find that the DMA bandwidth within Fastmove saturates when concurrency reaches 4 threads, exactly the same as the Optane PM experiments. This is because both the emulated NVM and Optane based experiments make use of I/OAT DMA, and under both cases, DMA bandwidth capacity is lower than Optane PM and emulated NVM.

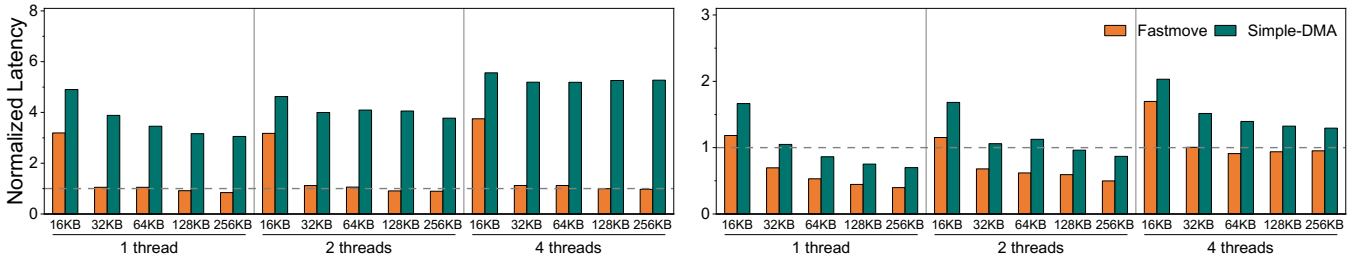
#### 6.4.2 DDIO Impacts

Enabling DDIO introduces no impact on the CPU-only baseline. However, DDIO affects DMA reads and writes differently. To unify our settings, we chose to turn DDIO off for our evaluation. With DDIO enabled, MySQL-TPCC-Fastmove outperforms the CPU-only baseline by 5.4%, but performs 15% worse than the DDIO-disabled counterpart. Unlike this, we observe no differences for GraphWalker's three algorithms, when switching on/off DDIO. This is because DDIO makes DMA writes slower and thus, does not affect GraphWalker whose workload is read-only.

## 7 Related Work

**I/OAT usage.** Previous studies have used I/OAT to offload memcpy operations that move data from DRAM to DRAM [46,47] as well as to improve network bandwidth with lower CPU utilization in data center environments [25, 48]. Unlike these, our study stands to speed up data movement between DRAM and NVM, where the interaction between I/OAT and hybrid memory architectures is more complex and its acceleration demands careful system design. Most recent work have included I/OAT as a minor optimization for data movement in NVM-based systems with special purposes ranging from log replication [7] to memory migration [41].

Unlike them, Fastmove is a general system to make use of on-chip DMA to address the inefficiencies (e.g., lower bandwidth or extra CPU overhead) introduced by CPU-only accesses to NVM for bulk, storage-facing I/Os, which has




(a) Local read (b) Local write

Figure 12: The latency comparison between Fastmove and Simple-DMA, with 1,2,4-threaded FIO workloads , normalized to the latencies of CPU-only memory copying when deploying NOVA on in-kernel NVM-emulator.

been observed to be a critical performance limitation in combined use of Optane PM and Intel processors. These systems can take advantage of Fastmove with little effort. Kalia et al. [21] present a number of optimizations for efficient remote NVM accesses via network, which includes an initial attempt to use I/OAT to improve single-core RPC performance for bulk remote NVM writes. This work is orthogonal to ours.

**NVM-related studies and systems.** There is a large body of work focusing on the analysis of the basic performance characteristics of using NVM [12, 14, 51, 55, 56]. The rich findings from these studies have spawned numerous studies for re-designing scalable and high performance data structures [24, 28, 50], file systems [19, 29, 53, 57] and key-value stores [9, 27]. Our work extends the existing study by incorporating the interaction between NVM and DMA, and complements the prior NVM-based systems as they can benefit from either the general design or the real implementation of Fastmove to alleviate data stalls. OdinFS [57] decouples application threads from the background NVM access threads and additionally parallelizes NVM accesses across sockets. Its NVM threads can benefit from Fastmove and its integration will be explored in the future.

**Tiered memory systems.** Fastmove handles more complex I/O patterns than those in tiered memory. In addition, Fastmove is implemented in the kernel with simple APIs. Therefore, Fastmove could be directly used in tiered memory systems. In fact, we have successfully adapted Nimble [54] to transparently use Fastmove through simple API replacement. However, through preliminary evaluations, we find that the DMA, in particular I/OAT, may not be a good option for improving page migration in tiered memory. This is because the DMA bandwidth is easily overwhelmed by the workload. Therefore, Fastmove does not deliver any significant improvement over Nimble-DMA [54], a Linux patch that adapts Nimble to use I/OAT.

**Zero-copy technologies.** Another line of work on PM attempts to move data management from kernel space to user space to eliminate data copies along the I/O path. For instance, the memory mapped file I/O (e.g., the mmap system call) is enabled such that users may access files in the same way as memory data [52]. However, mmap-based solutions may incur high overhead due to page faults [10, 26] and may have to have applications handle data persistence and reliability on their own [36, 38]. Yet another line of work leverages kernel by-pass I/O interfaces such as SPDK and PMDK [4] to avoid the use of the complicated OS I/O stack [44]. However, the performance gains come at the price of substantial effort for re-writing the I/O handling part of the target applications.

In contrast, our work demonstrates better applicability since there is no code change required to run existing applications atop Fastmove, as long as they use kernel file systems. Moreover, it is possible to extend our design to handle memory copy operations in user space, where these operations may have an even bigger impact on the overall performance compared to their counterparts in kernel space. This is because by bypassing the kernel, memory copying will contribute to a larger portion of the end-to-end access performance.

## 8 Conclusion

In this paper, we first study the DRAM-NVM data movement problem and then propose and implement Fastmove, a general engine that exploits the on-chip DMA technology. With a clean abstraction and transparent design, applications can use Fastmove via slightly-modified file systems with no further changes. Experimental results with industry-standard workloads on MySQL and popular random walk algorithms on GraphWalker highlight that Fastmove brings significant benefits such as peak throughput increase, execution time reduction, and CPU consumption savings.

## 9 Acknowledgment

We sincerely thank all anonymous reviewers for their insightful feedback and especially thank our shepherd Sanidhya Kashyap for his guidance in our camera-ready preparation. This work is supported in part by the National Natural Science Foundation of China under Grant No.: 62141216, 62172382 and 61832011, the USTC Research Funds of the Double First-Class Initiative under Grant No.: YD2150002006, and the Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00503, Researches on next generation memory-centric computing system architecture). Cheng Li is the corresponding author.

## References


[1] Filebench. https://github.com/filebench/filebench. [Online; accessed Jan-2023].

[2] Graph500. https://graph500.org/. [Online; accessed Jan-2023].

[3] MySQL. https://github.com/mysql. [Online; accessed Jan-2023].

[4] PMDK. https://github.com/pmem/pmdk. [Online; accessed Jan-2023].

[5] TPC Benchamrk C. http://tpc.org/tpcc/. [Online; accessed Jan-2023].

[6] Hiroyuki Akinaga and Hisashi Shima. Resistive random access memory (reram) based on metal oxides. *Proceedings of the IEEE*, 98(12):2237–2251, 2010.

[7] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. 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)*, pages 1011–1027, 2020.

[8] Jens Axboe. FIO. https://github.com/axboe/fio. [Online; accessed Jan-2023].

[9] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proc. VLDB Endow.*, 14(9):1544–1556, may 2021.

[10] Jungsik Choi, Jiwon Kim, and Hwansoo Han. Efficient memory mapped file i/o for in-memory file systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017.

[11] CXL Consortium. Compute Express Link: The Breakthrough CPU-to-Device Interconnect. https://www.computeexpresslink.org/, 2022. [Online; accessed Jan-2023].

[12] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. Maximizing persistent memory bandwidth utilization for olap workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD/PODS '21, page 339–351, New York, NY, USA, 2021.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[14] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. Understanding the idiosyncrasies of real persistent memory. *Proc. VLDB Endow.*, 14(4):626–639, December 2020.

[15] Frank T Hady, Annie Foong, Bryan Veal, and Dan Williams. Platform storage performance with 3d xpoint technology. *Proceedings of the IEEE*, 105(9):1822–1833, 2017.

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

[17] Dave Jiang. libnvdimm: add DMA supported blk-mq pmem driver. https://lore.kernel.org/linux-nvdimm/150412628764.69288.12074115435918322858.stgit@djiang5-desk3.ch.intel.com/#r. [Online; accessed Jan-2023].

[18] 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.

[19] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. Winefs: A hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 804–818, New York, NY, USA, 2021. Association for Computing Machinery.

[20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: Reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 494–508, New York, NY, USA, 2019.

[21] Anuj Kalia, David Andersen, and Michael Kaminsky. Challenges and solutions for fast remote persistent memory access. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 105–119, New York, NY, USA, 2020.

[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, 2018.

[23] Yoshihisa Kato, Yukihiro Kaneko, Hiroyuki Tanaka, Kazuhiro Kaibara, Shinzo Koyama, Kazunori Isogai, Takayoshi Yamada, and Yasuhiro Shimada. Overview and future challenge of ferroelectric random access memory technologies. *Japanese Journal of Applied Physics*, 46(4S):2157, 2007.

[24] Ana Khorguani, Thomas Ropars, and Noel De Palma. Respct: Fast checkpointing in non-volatile memory for multi-threaded applications. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 525–540, New York, NY, USA, 2022. Association for Computing Machinery.

[25] Jongyul Kim, Insu Jang, Waleed Reda, Jaeseong Im, Marco Canini, Dejan Kostić, Youngjin Kwon, Simon Peter, and Emmett Witchel. Linefs: Efficient smartnic offload of a distributed file system with pipeline parallelism. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 756–771, New York, NY, USA, 2021.

[26] Juno Kim, Yun Joon Soh, Joseph Izraelevitz, Jishen Zhao, and Steven Swanson. Subzero: Zero-copy io for persistent main memory file systems. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, page 1–8, New York, NY, USA, 2020.

[27] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young ri Choi, Alan Sussman, and Beomseok Nam. ListDB: Union of Write-Ahead logs and persistent SkipLists for incremental checkpointing on persistent memory. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 161–177, Carlsbad, CA, July 2022. USENIX Association.

[28] Wook-Hee Kim, R. Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. Pactree: A high performance persistent range index using pac guidelines. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 424–439, New York, NY, USA, 2021.

[29] Ruibin Li, Xiang Ren, Xu Zhao, Siwei He, Michael Stumm, and Ding Yuan. ctFS: Replacing file indexing with hardware memory translation through contiguous file allocation for persistent memory. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 35–50, Santa Clara, CA, February 2022. USENIX Association.

[30] Linux. Add support for NV-DIMMs to ext4. https://lwn.net/Articles/613384/. [Online; accessed Jan-2023].

[31] Linux. Device Mapper. https://www.kernel.org/doc/Documentation/device-mapper/. [Online; accessed Jan-2023].

[32] Linux. DMAEngine framework. https://www.kernel.org/doc/Documentation/driver-api/dmaengine/. [Online; accessed Jan-2023].

[33] Linux. xfs: DAX support. https://lwn.net/Articles/635514/. [Online; accessed Jan-2023].

[34] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. Octopus: an RDMA-enabled distributed persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, Santa Clara, CA, July 2017. USENIX Association.

[35] Maciej Maciejewski. How to emulate Persistent Memory. https://pmem.io/blog/2016/02/how-to-emulate-persistent-memory/, 2016. [Online; accessed Jan-2023].

[36] Ian Neal, Gefei Zuo, Eric Shiple, Tanvir Ahmed Khan, Youngjin Kwon, Simon Peter, and Baris Kasikci. Rethinking file mapping for persistent memory. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 97–111, February 2021.

[37] Philip Ng. Accelerating intra-host pvrdma storage traffic in a future dell amd server. talk at vmworld 2019, 2019. [Online; accessed Jan-2023].

[38] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, page 277–293, New York, NY, USA, 2021.

[39] Jonathan Prout. Expanding Beyond Limits With CXL-based Memory, 2022. [Online; accessed Jan-2023].

[40] Simone Raoux, Geoffrey W Burr, Matthew J Breitwisch, Charles T Rettner, Y-C Chen, Robert M Shelby, Martin Salinga, Daniel Krebs, S-H Chen, H-L Lung, et al. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, 2008.

[41] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021.

[42] Thomas Rueckes. High density, high reliability carbon nanotube nram. In *Flash Memory Summit*, 2011.

[43] Stackoverflow. Why are SIMD instructions not used in kernel? https://stackoverflow.com/questions/46677676/why-are-simd-instructions-not-used-in-kernel, 2022. [Online; accessed Jan-2023].

[44] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 431–445, Carlsbad, CA, July 2022. USENIX Association.

[45] AA Tulapurkar, Y Suzuki, A Fukushima, H Kubota, H Maehara, K Tsunekawa, DD Djayaprawira, N Watanabe, and S Yuasa. Spin-torque diode effect in magnetic tunnel junctions. *Nature*, 438(7066):339–342, 2005.

[46] K. Vaidyanathan, L. Chai, W. Huang, and D. K. Panda. Efficient asynchronous memory copy operations on multi-core systems and i/oat. In *2007 IEEE International Conference on Cluster Computing*, pages 159–168, 2007.

[47] K. Vaidyanathan, W. Huang, L. Chai, and D. K. Panda. Designing efficient asynchronous memory operations using hardware copy engine: A case study with i/oat. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8, 2007.

[48] Karthikeyan Vaidyanathan and Dhabaleswar K Panda. Benefits of i/o acceleration technology (i/oat) in clusters. In *2007 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 220–229. IEEE, 2007.

[49] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. Graphwalker: An i/o-efficient and resource-friendly graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 559–571, 2020.

[50] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and efficient multi-tenant persistent memory caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 1–16, Santa Clara, CA, February 2022. USENIX Association.

[51] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. Finding and fixing performance pathologies in persistent memory software stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 427–439, New York, NY, USA, 2019.

[53] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016.

[54] Zi Yan. Accelerate page migration and use memcg for PMEM management. https://lwn.net/Articles/784925/. [Online; accessed Jan-2023].

[55] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020.

[56] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. $MT^2$: Memory bandwidth regulation on hybrid NVM/DRAM platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, Santa Clara, CA, February 2022. USENIX Association.

[57] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. ODINFS: Scaling PM performance with opportunistic delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 179–193, Carlsbad, CA, July 2022. USENIX Association.