PLM$^{\text{light}}$: Emulating Predictable Latency Mode in Regular SSDs

Tanaya Roy*, Jit Gupta*, Krishna Kant*, Amitangshu Pal†, Dave Minturn‡
*Computer and Information Sciences, Temple University, Philadelphia, PA 19122, USA
†Computer Science and Engineering, Indian Institute of Technology Kanpur, Kanpur, India
‡Intel Corp, Hillsboro, OR 97291, USA

***Abstract*—The interactive web applications increasingly demand an end-to-end latency that is not only low on the average but also is "deterministic" in that they avoid long tails. Storage systems today largely keep data in SSDs, but SSDs are known to have unpredictable latencies due to background activities such as garbage collection. The recent NVMe access protocol proposes a Predictable Latency Mode (PLM) which allows the SSD to cycle between deterministic window (DTWin) and nondeterministic window (NDWin) periods, with background activities largely pushed to the latter. However, this means that the number of read and write IOs during DTWin period is limited and need to be managed properly. Another challenge is that to date no real SSDs are available in the market with this feature. In this paper, we explore the possibility of emulating the PLM feature in regular SSDs using Intel Optane that does provide a rather deterministic access latency. In particular, we propose a write I/O friendly PLM$^{\text{light}}$coordinator (PLMLC) that buffers writes in Optane and sends them to SSD during the NDWin-like period and also intelligently manages the limited number of IOs possible during DTWin-like period. The coordinator is designed to handle requests from multiple hosts that may access shared data on the SSD and may have different QoS requirements in terms of latencies. The results show that PLMLC improves the 99%-ile tail latency by 5.8x even without any sophisticated traffic estimation procedures.**

***Index Terms*—SSD, NVMe, Tail Latency, QoS, Predictable Latency Mode (PLM), Intel Optane**

I. Introduction

Storage systems form a crucial component of data center infrastructure and substantially influence the performance of the applications. Furthermore, nearly all of the storage resides away from the hosts in "storage servers," which means that network is an indispensable part of any storage access in current days. The emerging interactive applications are increasingly data-centric and require a low average storage access latency and one that does not vary a lot from request to request [1] [2] [3]. The low variation in latency is popularly expressed as a requirement of "deterministic" latency, but it means that the length and mass in the latency tail are tightly controlled.

The enterprise's ongoing replacement of hard drives (HDDs) by NVMe based solid-state drives (SSDs) has enabled high storage throughput and low storage access latency. Unfortunately, SSDs also need to internally carry out a complex array of background activities to create the impression of a simple read/write device. This simplicity is achieved by the Flash Translation Layer (FTL) that manages out-of-place writes, address translation, wear-leveling, and garbage collection [4]–[9]. These *background activities* can occasionally increase the IO latencies substantially, even to the level of several milliseconds [10].

978-1-6654-9550-9/21/$31.00 ©2021 IEEE

The storage access latency consists of several components including the device access latency, queuing latency for the device, network latency for the host to get to the storage server hosting the storage device, and host side latency. In this paper, our concern is only the device access latency, which often is a significant piece of the overall latency. Thus the notion of "determinism" only applies to this. The current version of NVMe protocol provides a feature known as predictable latency mode (PLM) to manage the latency caused by background activities by essentially limiting those activities to time windows known as NDWin (Non-deterministic windows) while no such operations are performed during the periods between NDwin periods, known as DTWin (deterministc windows). Now if we have two (or more) copies of data on different SSDs which never are simultaneously in NDWin mode, it is possible to provide deterministic service to the low latency class of jobs by serving them during DTwin periods.

Unfortunately, there are currently no available SSDs that implement this feature, and even when such SSDs become available, we have the question of providing some level of determinism to SSDs that don't have this feature. We tackle this in this paper by defining a mechnism known as PLM$^{\text{light}}$. Although the background operations may occur at random times in a regular SSD, we attempt to reduce their impact by bunching up the writes and not doing them during DTWin period. (The DTWin/NDWin periods are now artificially imposed durations explained later). The writes are held in a nonvolatile memory buffer that has rather deterministic access times (i.e., Intel Optane SSD considered) during the DTWin periods, and flushed to the SSD during the NDWin period. The write flush to SSD may trigger some garbage collection activity, which we expect to happen in the NDWin period. The objective then is to examine to what extent we can achieve deterministic read latencies and low write latencies with this arrangement.

We evaluate such a mechanism in a multi-host environment where it becomes necessary to coordinate hosts' access so that each host stream can be given the desired QoS treatment. In

particular, we assume that each host's IO stream belongs to one of the predefined set of QoS classes, and the coordinator, which we call PLM$^{\text{light}}$Coordinator (PLMLC), assigns how much of the IO capacity of a DTWin period each class can use. Any excess IO operations must be done in the NDWin period, where the latency is no longer deterministic. We show that PLMLC can help in achieving low latency and provides QoS differentiation as well. Although the overall performance is not as good as can be achieved via a real implementation of PLM, we note that we can achieve these results with widely available technologies and thus the solution can be easily used by existing storage systems without any need to replace the existing devices.

The outline of the rest of the paper is as follows. Section II discusses the background and the related work. Section III discusses the methodology we have employed to achieve deterministic latency in shared environment followed by Section IV which talks about the workloads used and the experimental evaluation. Finally, section V concludes the paper.

II. Background and Related Work

Traditionally, the SSD firmware controls these background activities with little regard to the ongoing IO operations. Certain implementation choices such as incremental garbage collection or incremental wear-leveling can reduce the disruptive impact of these operations [11]–[14]. However, to eliminate them, it is necessary to either put the scheduling in the host's hands (so that the host can schedule them as appropriate) or provide a mechanism by which the applications needing determinism can always have their requests processed without interference. The former is supported by the emerging concept of ZNS (Zoned Name Space) [15] where the SSD is divided into several "zones" each of which is managed independently as a log-structured "device," i.e., data can only be appended until the zone fills up. The host controls the garbage collection of zones and can be avoided during IOs of applications requiring deterministic latency. However, the additional burden placed on the host for managing zones is a downside, making its widespread adoption difficult.

NVMe 1.4 specification (NVMe v1.4) proposes another mechanism to achieve deterministic latency without burdening the hosts. However, it has the disadvantage of requiring the data to be replicated on two (or more) identical devices, similar to but not identical to RAID 1. It is known as the Predictable Latency Mode (PLM), and each SSD in the RAID-1 like pair cycles through time windows known as "DTWin" (deterministic window) and "NDWin" (non-deterministic window) in a way that at least one of them is always in the DTWin mode. In the DTWin mode, background operations are held back so that the device is dedicated to serving the requests.

NVMe1.4 standard does not offer clear guidance as to how write I/O should be handled. This standard suggests that the writes during the DTWin period could be absorbed by an NVRAM buffer inside the SSD and then transferred to SSD during the NDWin period. Such an arrangement can work nicely for read-only traffic since the data can be read from whichever SSD is in DTWin mode; however, writes introduce three complications: (a) the writes must be reflected on both copies, (b) writes could trigger background activities (e.g., garbage collection) and (c) a write cannot be postponed as that would result in serving stale data. However, this brings in several issues regarding the sizing of NVRAM buffer and corresponding cost implications and the impact of buffer overflow (which would require a forced transition from DTWin to NDWin mode). The NVMe1.4 mechanism does not address the issue of data being shared among multiple hosts, such as in the case of a distributed database system. Moreover, no commercially available SSD has implemented this feature so far; therefore, it is impossible to evaluate it directly.

In the work [16], the read IO management is explored in a shared environment by modifying an extremely detailed SSD simulator called MQSim [7]. The designed coordinator synchronizes hosts (that itself looks like a host to the device) to observe the IO use of each host and distributes the IO capability during the DTwin period among various hosts. We also introduced the notion of QoS and showed how the IO count allocation could be done to respect the QoS requirements.

A. SSD Structure and Management Activities

Fig.1 shows the details of NVM storage structure. An NVMe SSD is partitioned into one or more "NVM subsystems" which can be managed independently. An NVM subsystem comprises of one or more NVMe controllers, one or more NVM subsystem ports, non-volatile storage medium and an interface between the controllers and the storage medium. Therefore, an SSD can be deemed as an NVM subsystem. Each NVM subsystem may consist of one or more Endurance Groups (EG) to localize garbage collection. Each EG could contain a number of "NVM Sets" (NVMS), and each of those can consist of one or more Namespace (NS).

The *NVMe Storage Controller* (NSC) helps to connect a host with the NVM storage. An SSD can be accessed by a host locally as well as remotely. For remote access, a protocol called NVMe-oF (NVMe over fabric) has been defined that essentially extends the NVMe commands to be ported from host to the target and executed remotely [17].

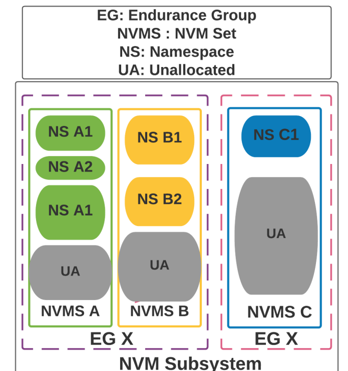


Fig. 1: NVM Storage Hierarchy

The PLM is operated at the granularity of NVM-sets, which cycles between the aforementioned DTWin and NDWin as in Fig 2 such that all background activities are performed during the NDWin period. The **DTWin budget** can be represented by three attributes: (a) a predefined time limit, (b) a predefined limit on the number of reads, and (c) a predefined limit on the number of the writes

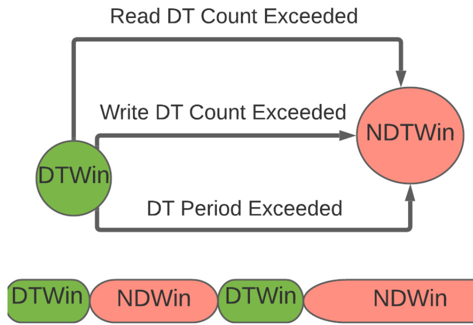


Fig. 2: PLM Mechanism

that can be performed on the NVM-set. Both (b) and (c) are collectively termed as *Deterministic Counts (DC)* in this paper. The NVM-set may transition to NDWin if either of these limits is exceeded. There are various DTWin attributes introduced in NVMe v1.4 such as DTWin read typical (C_{RD}), DTWin Write typical (C_{WR}), and DTWin time maximum (T_{DW}). We consider the parameters C_{RD} and C_{WR} as DTWin read deterministic count ($read_DC$) and write deterministic count ($write_DC$) respectively.

B. PLM Feature and Its Functioning

The PLM feature proposed in [15] is primarily designed to cater to reads since the latencies for reads are generally more critical than for the writes. To achieve the deterministic latency, the host needs to obey the predictable latency operating rules. As stated earlier, to provide continuous access to DTWin mode to an application, we need multiple (two or more) NVM-sets that are identical and contain copies of the data. If all NVM-set copies are never simultaneously placed in the NDWin state, an application can read data in the DTWin mode any time. While the required duplication is expensive, it is needed only for the data accesses that must be deterministic; one copy suffices for all other data. Also, if multiple copies are maintained for resilience reasons, they can be used to provide the deterministic service as well. It is also worth noting that a traditional RAID1 (mirroring) does not provide the same functionality as PLM since the reads could occur in RAID1 while the SSD is doing the background operations.

The NVM-set may transition to the NDWin before exhausting its DTWin budget – this happens if there is an emergency management operation required such as an emergency garbage collection being triggered. The amount of time spent by the NVM-set in the NDWin state depends on the necessary background operations for management activity (and the flushing of writes accumulated in the buffer.)

A remote host can setup a connection via NSC. The NSC advertises the DTWin related status for each NVM-set into a "log-page". The log-page is accessible to the host using NVMe commands and therefore the DTWin information per NVM-set. The host defines the beginning of DTWin and NDWin periods by making requests to the NSC using the "set-feature" command. Since a host could use multiple NVM-sets simultaneously, they could be at different points in their DTWin/NDWin cycles. Therefore, the host must influence DTWin/NDWin duration to synchronize multiple copies of NVM-sets. PLM's current specification puts the regular transition between DTWin and NDWin control with the host (if the log-page advertised status is respected) and the forced transition with the NSC (if the host does not respect the operating rules).

C. Coordinator Background

One of the initial issues we encountered in our study of the PLM feature was the associated problems while extending it to a multi-host environment. In such an environment a resource allocation module is required along with the existence of usual NSC capabilities. This module can be a part of the NSC itself; however, running separate workloads pertaining to the different hosts can create a bottleneck at the NSC. This overhead brings forth the need to break this module into sub-modules and place it outside the NSC in a suitable setting i.e. the storage server. This issue is explored in [16] by introducing the concept of a PLM Coordinator (PLMC), which acts like the aforementioned DTWin budget allocation entity that distributes $read_DC$/T_{DR}. It interacts with the hosts and allocates for them their $read_DC$ (depending on their QoS requirements) at the start of each DTWin period. The PLMC does not participate in the data path of the I/O requests, however, it is also possible to place the PLMC on the host side as another host requesting for the shared NVMe Set.

D. Hybrid Storage Model

Hybrid storage model is quite popular this days for distinctive purposes. In [18] surveys with different set of storage technologies choices with their usefulness and pitfalls. The hybrid storage model with Optane memory, used in the same level with SSD or as a different level in the storage hierarchy, to enhance the performance and endurance of the overall storage system is explored by multiple researchers [19]–[21]. In [19], they have valuable findings to use for building emerging hybrid SSD controllers. They have observed, it is suitable to place small files to be placed in the Optane and large files on SSDs. The Optane is used to enhance the endurance of the overall storage system in [20]. [21] caches datablock based on the access eligibility of a block from SSD.

In our work, we have considered Optane in the same level as SSD to emulate the PLM feature. Therefore, there are no existing works with which we can compare the realized deterministic latency.

III. Achieving Deterministic Latency in Shared Environment

As stated earlier, PLMC is capable of distributing $read_DC$ among the active hosts based on the application QoS. However, it doesn't consider write IOs. The shared environment becomes interesting when write intensive workloads are in play simultaneously and share the same working set. To assure consistency it is required to execute the write IO when requested. However, inexpert handling of write IO can consume the

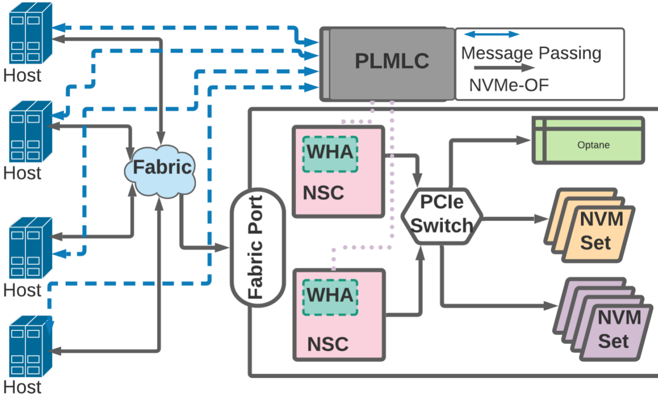


Fig. 3: An Illustration of PLM$^{\text{light}}$ Architecture

device $write_DC$ and therefore can force DTWin to NDWin transition. This can result in a long spike in latency for latency sensitive applications. In this paper, we have proposed a coordinator - PLM$^{\text{light}}$Coordinator (PLMLC), to manage both read IO and the write IO in regular SSDs while embodying the features from PLM. Thus guaranteeing the low latency for high QoS applications.

A. Proposed PLMLC in a Shared Environment

We design the proposed PLM$^{\text{light}}$to distribute the $read_DC$ among multiple hosts running different workloads that share the storage space while also handling write IOs. PLM$^{\text{light}}$has two sub-modules, the central coordinator agent i.e. PLMLC along with a *Write Handling Agent* (WHA) in each server. The PLMLC is placed on the control path, outside the storage controller. It acts as a host and communicates with the storage server using NVMe-OF. Therefore, it can collect the status of the PLM parameters from the NSC using NVMe commands. Also, it can collect the traffic experienced by each host via message passing to allocate $read_DC$ for each host. The write IO needs to be executed immediately to maintain consistency and reliability. To control the $write_DC$ usage of each NVM set, limited write IOs are performed on SSD. The excess write IOs need to be relocated in order to achieve deterministic latency. Therefore, the WHA needs to be placed in the storage server along with the NSC to direct the location of the write IO execution. In the proposed PLM$^{\text{light}}$, we need an additional device (which reports more deterministic write latencies) along with the regular SSDs to complete the aforementioned excess write IO, thus creating a *Hybrid Storage Model*.

In this paper, we have considered Intel Optane SSD to accomplish these excess writes. The architecture of PLM$^{\text{light}}$is shown in the Figure 3. PLMLC allocates $read_DC$ to each host actively accessing the NVMe target, consisting of two or more NVM subsystem(s), i.e SSD(s), and Optane as the additional device. It also maintains a consolidated $write_DC$ pertaining to the device so as to ensure the device's seamless transition to NDWin when $write_DC$ is exhausted. Each subsystem can have one or more NSCs that control a number of NVM-sets. The hosts can communicate with PLMLC using conventional NVMe-OF protocol [17], [22] and access each NVM-set's log-page to collect read DC information. It is assumed that each host is reliable to convey its usage to PLMLC and respect the allocated limit. The PLMLC needs to learn the workload traffic characteristic to estimate the required DC. It allocates the $read_DC$ to each host based on the $read_DC$ estimation and the QoS class of the applications running on each hosts. Unlike PLMLC, there is no separate communication needed between hosts and WHA. The WHA decides the write IO execution device depending on the device characteristics.

In general, hosts with similar traffic characteristics will experience the same underlying device access latency (exclusive of any queuing or other latencies) in the absence of management operations. Therefore, the difference will be highlighted in tail latency. The end-to-end (E2E) latency will consist of different components, such as: (a) host-side IO dispatch and IO completion latency, (b) network transit latency, (c) NVMe queuing and queue handling latency, and (d) device access latency. In reality, the QoS class definition would cover the end-to-end latency, and must be split into allowable latencies for each of these four components. For example, if TL is the overall desired 99% latency, we can split it into four pieces $TL_i, i = 1..4$ with $\sum_{i=1}^{4} = TL$. This paper is only focused on (d) following such a split.

B. Host Traffic Estimation by PLMLC

The PLMLC module of PLM$^{\text{light}}$needs the estimated $read_DC$ of each host and distributes the available $read_DC$ considering the application's QoS class. The estimation could be done either by each host individually and then communicated to PLMLC, or the PLMLC can do the estimation itself. The former is more cumbersome and perhaps not desirable since it adds extra burden on the hosts and could be subject to misbehavior if a host's estimation is faulty (e.g., a deliberate overestimation supplied by the application). However, the host can potentially use information that is not available to PLMLC, including perhaps the precise number of reads/writes to be performed during a DTWin.

The PLMLC can easily separate the requests from each host and do a statistical estimation of the traffic. However, the key difficulty in such an estimation is that the storage traffic is generally very bursty and nonstationary. Thus the standard quasi-stationary time-series prediction models, such as ARMA, Kalman-filters, etc. do not provide very good predictions [23]. In fact, our experiments with all these algorithms (not reported here) confirmed that these well-established methods did not provide any significant advantage over a simple exponential smoothing-based traffic estimation. In theory, more sophisticated neural net models could do better by capturing the patterns in the workload, however, this method has two key hurdles: (a) need for very long traces for training the model, and (b) need to retrain when workloads/applications change. Our results show that basic exponential smoothing provides satisfactory results in spite of its simplicity.

Let $\mathcal{R}_j^{(m)}(n)$ denote the measured read request count (traffic) at the n-th DTWin for the j-th class and $\mathcal{R}_j^{(p)}(n)$ their

smoothed estimated in the same period, with $0 < \zeta < 1$ as the smoothing constant. Then,

$$\mathcal{R}_j^{(p)}(n+1) = \zeta\mathcal{R}_j^{(m)}(n) + (1-\zeta)\mathcal{R}_j^{(p)}(n) \quad (1)$$

Where ζ is chosen experimentally as 0.5 for the entirety of each experiment.

C. Deterministic Count Allocations by PLMLC

For the DC allocation, we have closely followed our previous work in [16] where we considered k hosts running different workloads with different QoS classes represented as $Q_1, Q_2, ..., Q_k$. The n-th DTWin duration is termed $W_d(n)$ which is followed by the subsequent NDWin of duration $W_{nd}(n)$. The entirety of these two periods is collectively termed as the *scheduling period* $W(n)$, where

$$W(n) = W_d(n) + W_{nd}(n) \quad (2)$$

The write request behavior during a preceding $W_d(n)$ determines the duration of a succeeding $W_{nd}(n)$ as the writes may trigger background activities which in turn extend said $W_{nd}(n)$. During each $W(n)$ we need to allocate the total DC among the k hosts so as to establish differentiated treatment for all of $Q_1, Q_2, ..., Q_k$ classes and to also make sure that DC allocated is not wasted (i.e. a certain does not use its allocated DC)

1) Read DC allocation: We term the total available $read_DC$ during a $W_d(n)$ as C. If the predicted traffic results in a total required $read_DC$ is below C, we then allocate the estimated value itself. However, if the predicted value exceeds C, we follow a fixed ratio style of allocation for the excess required $read_DC$, which is as explained below.

Fixed Ratio Allocation Policy: In this allocation policy, the residual required $read_DC$ is split among various hosts based on a predefined set of ratios r_j corresponding to the Q different classes. We further divide the allocation policy into two different styles so as to observe how it affects the QoS differentiation. This is done on the basis of how much minimum $read_DC$ is guaranteed to each host. They are as follows -

- Equal Base (EB): In every DTWin, each QoS host starts with the same minimum $read_DC$. Though, additionally, they may get more $read_DC$s based on their traffic characteristics and QoS class.
- Prioritized Base (PB): At the beginning of each DTWin, each host is assigned $read_DC$s based on their QoS class. Additionally, more $read_DC$s are also allocated based on the QoS class, similar to *EM*.

In contrast to [16], we have considered only the fixed ratio policy as the other method mentioned in [16] (Strict Priority) and Fixed Ratio tend to perform similarly in a variety of situations.

2) Write DC management: We can distribute the $read_DC$ considering the traffic intensity and QoS class of the workload. In that case, we can force low QoS class workloads to wait and perform their execution during NDWin, when allocated read DC is exhausted before the DTWin ends. In the case of $write_DC$, we cannot stall the write operations until DTWin ends as the working set is shared among all the hosts. This action will violate consistency. Therefore writes need to be performed immediately. This is also the reason why we refrained from differentiating between QoS classes for allocating $write_DC$. To mitigate the premature end of DTWin, we will reduce the number of writes going to the SSD. Here we consider a policy where small writes ($4K$ to $64K$ IO size in our case) are done in Optane and large writes (larger than $64K$) in SSD.

Workload	High	Medium	Low
$WorkloadI$	421.6	424.7	426.3
$WorkloadII$	383	397.2	396
$WorkloadIII$	1161.6	1163.2	1160.4
$WorkloadIV$	1117.3	1131.1	1132.6

TABLE I: Average IO in KB/DTWin

IV. Evaluation of the Proposed PLMLC

A. Workloads and Configurations Used

To evaluate our mechanism, we use the widely accepted FIO benchmarking tool to create workloads that can stress test PLMLC [24]. We ran FIO with four different configurations which we denoted as I-IV. These workloads were spawned into three different processes with each process pinned to a different CPU. The CPUs are considered as three hosts in our description. The workload configurations are as follows.

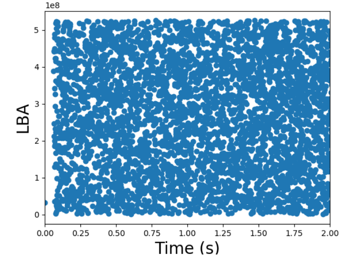


Fig. 6: Snapshot of W-I

1) **Workload I and II:** The first two workloads have a read-write ratio of 30:70 and 70:30 respectively. Both of these workloads have a request size of 4KB. This tests PLMLC 's performance for single block requests.
2) **Workload III and IV:** Here, the request size for these two workloads range from 4KB to 128KB, but the same 30:70 and 70:30 read/write ratios are maintained. This tests PLMLC's performance for variable and larger request sizes.

Due to the pervasive use of caching and prefetching throughout the storage hierarchy, we considered primarily 100% random workloads. This presents the most challenging workload for PLMLC. We can observe this random behavior in Figs 6 and 7 and can also observe how the variable request size of Workload IV ensures that the rate of requests is smaller as compared to that for Workload I. Both these figures show a two-second snapshot of workloads I and IV.

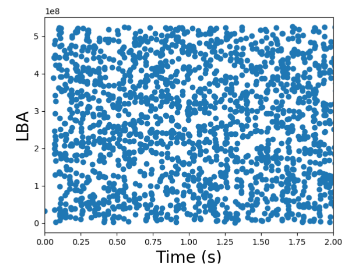
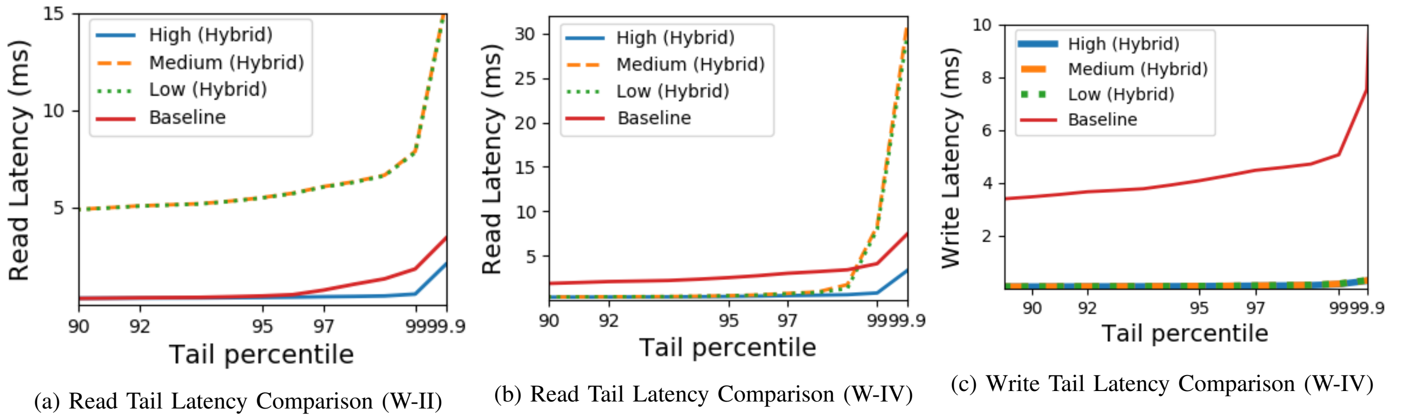


Fig. 7: Snapshot of W-IV



(a) Read Tail Latency Comparison (W-II)

(b) Read Tail Latency Comparison (W-IV)

(c) Write Tail Latency Comparison (W-IV)

Fig. 4: Read and Write Tail Latency Comparison with Baseline for Workloads II and IV

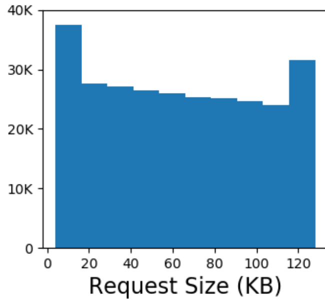


Fig. 5: Histogram depicting request size distribution in W-III

We have used the Linux applications *blktrace* and *blkparse* to capture the replayable trace pertaining to each of the workloads generated by FIO. The resultant trace contained four actions (in the given order) pertaining to each request -

1) **Q / Queue Request:** This action notes the intent to queue the request at the given location. No request actually exists yet.
2) **G / Get Request:** This action corresponds to allocating a *struct request* container, which is a mandatory step to send any type of request to the block device.
3) **D / Issue Request:** As the name suggests, this action issues the created existing request to the driver.
4) **C / Completed Request:** Finally, this notifies the completion of the request.

For our evaluation, we have only considered the action **D** as that is the only action of significance to us. This is because every other action is dependant on how the application itself behaves, for example, the request may be completed at a different time in FIO compared to PLMLC. The final trimmed traces (with a single action pertaining to a single request) were used for our experimentation.

B. Evaluation Metrics

In this paper, our goal is to achieve low latency with minimum variation. Therefore, measuring and comparing tail latencies, along with the average latencies, are useful here.

- Read Tail Latency: For each of the considered workloads, read latencies are measured and tail latency at different %-tile, viz. 90%-tile, 95%-tile, 99%-tile, 99.9%, are reported. These values represents the maximum latency value, for the fastest 90%, 95%, 99% and 99.9% of read IO requests respectively.
- Write Tail Latency: Similar to the read latency, write latencies are measured and tail latencies are reported.

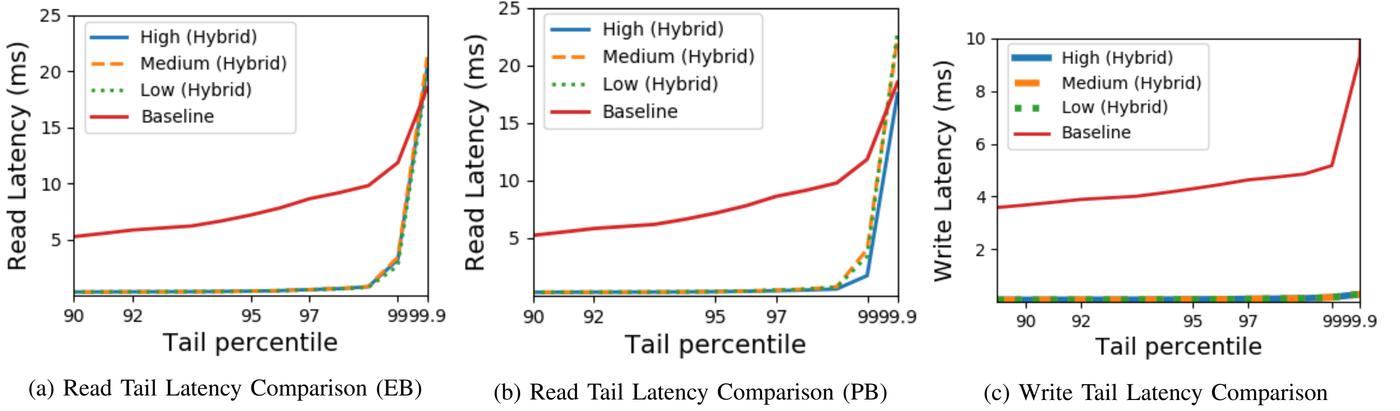
C. Experimental Setup

In this paper, to achieve our goal of deterministic latency, we have emulated the NVMe 1.4 proposed PLM using workloads containing both reads and writes in a distributed shared environment. We have considered two identical SSDs, NVMe SSD 970 EVO Plus, as NVM-sets. The Optane memory on PCIe is considered as the write buffer to these NVM sets. As there is no common driver application to manage both SSDs and Optane memory, we have implemented an application that manages both SSDs and Optane in a Linux environment, Ubuntu 20.04. We have turned off the SSDs internal write cache and Linux page cache to reduce the effect of caching mechanism in the experienced latency of an IO request.

D. Evaluation Results

For evaluation, we have considered the above four workloads(I-IV). Each of these workload types runs on three hosts with different QoS class requirements. We have performed a baseline experiment with three hosts running simultaneously for each workload on SSD without any QoS-based treatment. As there is no differential treatment for the three hosts in the Baseline scenario, all three exhibit close to identical behavior with negligible difference. Hence we have only depicted one of the hosts in our figures as the *Baseline* comparison. We compared our $\text{PLM}^{\text{light}}$ performance for all the workloads against the baseline. We performed the $\text{PLM}^{\text{light}}$ performance tests using the fixed ratio policy with EB and PB. For each of the workloads, we considered three hosts in the *Hybrid Storage Model* (as mentioned in Section IIIA) with different QoS classes (high, medium and low) running together.

For each of the workloads, we considered consistent configurations depending on the workload characteristics. The available $read_DC$ and $write_DC$ for every DTWin are the same and considered as 80% of the average $read$ and $write$ IOs per DTWin. The DTwin period is considered as 1



(a) Read Tail Latency Comparison (EB)

(b) Read Tail Latency Comparison (PB)

(c) Write Tail Latency Comparison

Fig. 8: Read and Write Tail Latency Comparison with Baseline for Workload III

millisecond. Therefore, 80% of the $read$ and $write$ IOs will be served in 1 millisecond. We want to examine how efficiently PLM$^{\text{light}}$can distribute this DTWin IO capacity among three classes. The PLM$^{\text{light}}$performance tests are done considering both EB and PB, where total base $read_DC$ collectively assigned for high, medium and low are 75% of the average $read$. In the case of EB, the total base $read_DC$ is shared among three QoS equally as mentioned in Section III-C. In contrast, total base $read_DC$ is shared between QoS classes as 50%, 30% and 20% for high, medium and low respectively. Therefore, for a high IO intensive application, all QoS class have the possibility to achieve the required amount of DC to experience low read latency. PLM$^{\text{light}}$doesn't distribute $write_DC$ among the QoS classes. All QoS classes share the $write_DC$ on a first-come-first-serve basis.

1) Write-intensive IO Workload: Write intensive workloads (Workloads I and III) running on SSD can yield a few milliseconds read latency as shown in Figs 8a and 8b and considered as baseline. Therefore, it is reasonable to compare our PLM$^{\text{light}}$considering the *Hybrid Storage Model* with the baseline. Baseline has millisecond-level tail latency from 90 percentile and beyond. Whereas, PLM$^{\text{light}}$has microsecond-level tail latency at 99%-tile. The tail touches the millisecond-level latency at 99.9%-tile, which is not different than baseline. At 99%-tile, the High QoS host under PLM$^{\text{light}}$in Workload III improves 2.7x and 5.8x than baseline for EM and PM respectively. With these workload characteristics and considered DC configuration, there is not much differentiated service provided with EM policy. However, for PM policy, PLM$^{\text{light}}$provides significant differentiated service. Therefore, the High QoS host performs better by 0.5x compared to the Low and Medium QoS host for 99%-tile tail latency. The write latency also significantly improves with PLM$^{\text{light}}$as shown in Fig 8c. It happens as Optane consumes a large portion of writes. The IO size distribution of this workload is represented in Fig 5. All writes with IO size greater than 64K are consumed by the SSD. Otherwise, it is persisted in the Optane. We cannot see differentiated service for write latency among the three QoS classes as $write_{DC}$'s are used collectively.

Further, we wanted to explore the write intensive workload performance with PLM$^{\text{light}}$, where all writes are small (4KB) i.e. Workload I. Fig 9 shows that PLM$^{\text{light}}$can outperform baseline for High QoS. It improves close to 0.5x at 99.9%-tile. However, the differentiated service results in higher latency for low and medium QoS than baseline in this Workload. Moreover, when we consider the writes, the latency for all the QoS classes stays at microsecond values as shown in Fig 10. This happens as writes are all consumed by the Optane buffer and transferred to SSD during the NDWin period.

2) Read-intensive IO Workload: Although PLM$^{\text{light}}$is primarily focused on write intensive workloads, we wanted to conduct performance tests considering the read intensive workloads (Workloads II and IV) too. Fig 4a shows that high QoS read tail latency performs better than baseline by 70% at 99%-tile and 30% at 99.99%-tile with a hybrid model. The primary reason of this small improvement in read latency is the small portion of write data goes to Optane and results in a spike in SSD latency. Therefore, any amount of writes dissociated from SSD can yield better read tail latency. This finding can be made stronger with a performance test with significant IO size variations even in read intensive workload.

In this workload (Workload IV), we have variations in IO size, and a small fraction of writes with less than 64KB IO size persists in the Optane, while the rest go to the SSD. In Fig 4b, the high QoS outperforms the baseline read tail latency by 80% for 99%-tile and 55% for 99.9%-tile, which is almost similar to the performance achieved with the read-intensive small IO workload (Workload II). It is noticeable that low and medium QoS class read latency improves from the other read-intensive workload (Workload II) by avoiding write IO execution in SSD. Although, a small amount of write is present in the workload, persisting a portion of it in Optane improves the write tail latency significantly in PLM$^{\text{light}}$as compared to the baseline (shown in Fig 4c).

From the described result graphs we have noticed that the write-intensive low and medium workloads perform better than read-intensive workload settings for read latency. In case of write-intensive workload, fig 9 and fig 8 the considered read deterministic counts are sufficient for the read IO execution during DTWin and therefore, helps medium and low

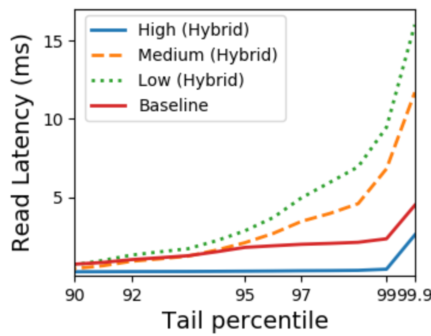


Fig. 9: Read Tail Latency Comparison for W-I

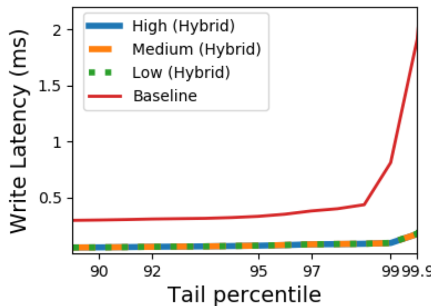


Fig. 10: Write Tail Latency Comparison for W-I

workloads to achieve low latency. In case of read-intensive scenario, fig 4, we have considered a stressed situation, where device DCs are not enough for the ongoing read and write IO operations. Therefore, high is given highest preference and as an effect, the medium and low suffered significantly.

V. Conclusions and Future Work

In this paper we have considered a mechanism to enhance the determinism of SSD access latency by making use of DTWin/NDWin concepts proposed in NVMe1.4, coupled with the introduction of a more deterministic device in the overall architecture. Since no commercially available SSDs currently have yet implemented this feature, and there is a huge installed base of SSDs, our proposed mechanism instead attempts to increase determinism by artificially defining DTWin/NDWin periods such that minimal writes to SSDs are issued during the DTWin period. Instead, most writes are directed to an Intel Optane device that inherently provides deterministic latency. These writes are opportunistically transferred to SSD during NDWin periods. In this paper, we design a coordinator that can handle requests from multiple hosts to a shared device. The coordinator distributes the allowed IO counts to individual hosts based on their QoS requirements. A detailed evaluation shows that mechanism can substantially enhance the determinism and also reduce the write access latencies by close to 5.8x.

Acknowledgment

This research was supported by a grant from Intel Corporation.

References

[1] Y. Xu *et al.*, "Bobtail: Avoiding long tails in the cloud," in *USENIX NSDI*, 2013, pp. 329–341.

[2] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS operating systems review*, vol. 41, no. 6, pp. 205–220, 2007.

[3] J. Dean *et al.*, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[4] A. I. Alsalibi *et al.*, "A survey of techniques for architecting slc/mlc/tlc hybrid flash memory–based ssds," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 13, p. e4420, 2018.

[5] S. Kim *et al.*, "Analysis of potential risks for garbage collection and wear leveling interference in ftl-based nand flash memory," *Journal of The Korea Society of Computer and Information*, vol. 24, no. 3, pp. 1–9, 2019.

[6] Y. Cai *et al.*, "Error characterization, mitigation, and recovery in flash-memory-based solid-state drives," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1666–1704, 2017.

[7] A. Tavakkol *et al.*, "Mqsim: A framework for enabling realistic studies of modern multi-queue SSD devices," in *USENIX FAST*, 2018, pp. 49–66.

[8] J. Kim *et al.*, "Alleviating garbage collection interference through spatial separation in all flash arrays," in *USENIX ATC*, 2019, pp. 799–812.

[9] A. Tavakkol *et al.*, "Flin: Enabling fairness and enhancing performance in modern nvme solid state drives," in *ACM/IEEE ISCA*, 2018, pp. 397–410.

[10] https://www.intel.com/content/dam/www/public/us/en/documents /technology-briefs/low-latency-for-storage-intensive-workloads-tech-brief.pdf, 2020.

[11] S.-M. Huang *et al.*, "Providing slo compliance on nvme ssds through parallelism reservation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, no. 3, pp. 1–26, 2018.

[12] J. Guo *et al.*, "Parallelism and garbage collection aware i/o scheduler with improved ssd performance," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017, pp. 1184–1193.

[13] Y. Du *et al.*, "Enhancing ssd performance with ldpc-aware garbage collection," in *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2017, pp. 1–4.

[14] J. Do *et al.*, "Improving cpu i/o performance via ssd controller ftl support for batched writes," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, 2019, pp. 1–8.

[15] "Nvm express base specification, rev 1.4," https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf, June 2019.

[16] T. Roy *et al.*, "Managing ssd tail latency with plm," *Proc. of NAS*, Oct 2021.

[17] D. Minturn *et al.*, "Under the hood with nvme over fabrics," in *Ethernet Storage Forum. SNIA*, 2015.

[18] M. Hoseinzadeh, "A survey on tiering and caching in high-performance storage systems," *arXiv preprint arXiv:1904.11560*, 2019.

[19] H. Chen *et al.*, "An empirical study of hybrid ssd with optane and qlc flash," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*. IEEE, 2020, pp. 175–178.

[20] T. Roy *et al.*, "Enhancing endurance of ssd based high-performance storage systems using emerging nvm technologies," in *IEEE IPDPS Workshops*, 2020, pp. 1070–1079.

[21] J. Gunta *et al.*, "Fussycache: A caching mechanism for emerging storage hierarchies," in *2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2020, pp. 74–81.

[22] Z. Guz *et al.*, "Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation," in *ACM SYSTOR*, 2017, pp. 16:1–16:9.

[23] A. S. Weigend, *Time series prediction: forecasting the future and understanding the past*. Routledge, 2018.

[24] J. Axboe, "Fio," 2013. [Online]. Available: https://fio.readthedocs.io/en/latest/