Extending and Programming the NVMe I/O Determinism Interface for Flash Arrays

HUAICHENG LI, University of Chicago and Carnegie Mellon University,
MARTIN L. PUTRA, University of Chicago,
RONALD SHI, University of Chicago,
FADHIL I. KURNIA, University of Massachusetts Amherst,
XING LIN, NetApp,
JAEYOUNG DO, Microsoft Research,
ACHMAD IMAM KISTIJANTORO, Bandung Institute of Technology,
GREGORY R. GANGER, Carnegie Mellon University,
HARYADI S. GUNAWI, University of Chicago,

Predictable latency on flash storage is a long-pursuit goal, yet, unpredictability stays due to the unavoidable disturbance from many well-known SSD internal activities. To combat this issue, the recent NVMe IO Determinism (IOD) interface advocates host-level controls to SSD internal management tasks. While promising, challenges remain on how to exploit it for truly predictable performance.

We present IODA[1]*, an I/O deterministic flash array design built on top of small but powerful extensions to the IOD interface for easy deployment.* IODA *exploits data redundancy in the context of IOD for a strong latency predictability contract. In* IODA*, SSDs are expected to quickly fail an I/O on purpose to allow predictable I/Os through proactive data reconstruction. In the case of concurrent internal operations,* IODA *introduces busy remaining time exposure and predictable-latency-window formulation to guarantee predictable data reconstructions. Overall,* IODA *only adds 5 new fields to the NVMe interface and a small modification in the flash firmware, while keeping most of the complexity in the host OS. Our evaluation shows that* IODA *improves the 95–99.99th latencies by up to 75×.* IODA *is also the nearest to the ideal, no disturbance case compared to 7 state-of-the-art preemption, suspension, GC coordination, partitioning, tiny-tail flash controller, prediction, and proactive approaches.*

CCS Concepts: • **Computer systems organization → Firmware**; **Embedded hardware**; **Embedded software**; • **Information systems → Flash memory**; • **Hardware → Emerging interfaces**.

Additional Key Words and Phrases: Software/Hardware Co-Design, Predictable Latency, SSD

[1] An earlier version of the paper appeared at ACM SOSP'21 [1].

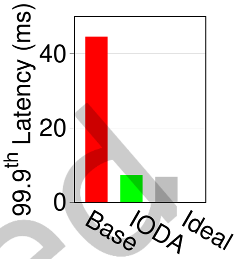
Authors' addresses: Huaicheng Li, University of Chicago and Carnegie Mellon University, ; Martin L. Putra, University of Chicago, ; Ronald Shi, University of Chicago, ; Fadhil I. Kurnia, University of Massachusetts Amherst, ; Xing Lin, NetApp, ; Jaeyoung Do, Microsoft Research, ; Achmad Imam Kistijantoro, Bandung Institute of Technology, ; Gregory R. Ganger, Carnegie Mellon University, ; Haryadi S. Gunawi, University of Chicago,

1553-3077/2022/11-ART $15.00
https://doi.org/10.1145/3568427

1 INTRODUCTION

Flash arrays are popular storage choices in data centers and they must address users' craving for low and predictable latencies [2–4]. Thus, many recent SSD products are released and evaluated not just on the average speed but the percentile latencies as well [5–8]. These all paint the reality that customers would like SSDs with *deterministic* latencies.

Deterministic latency, however, is hard to achieve because SSD performance is inherently non-deterministic due to the internal management activities such as the garbage collection (GC) process, wear leveling, and internal buffer flush [9–11]. These activities will inevitably trigger many background I/Os and disturb user requests. Notably, GC is a necessary path to overcome NAND Flash's inability for in-place overwrites. It involves time-consuming data movement to reclaim space and contend with user requests, thereby causing severe latency hiccups. As an illustration, the figure on the right shows the giant latency gap between the "Base" (with GC) and the "Ideal" (no GC) cases. Modern SSDs often resort to large over-provisioning space (*e.g.*, up to 50% of the SSD's raw NAND capacity) [12] to provide legroom for more efficienct background task processing, however, our profiling experiments on recent enterprise SSDs showed that GCs can still cause up to 60× latency increase (details omitted). This is unfortunately still an ongoing problem faced by the storage industry [13–15].



To tame the SSD performance challenges, there have been many efforts to evolve the device interfaces [16–18]. The Storage Interface Technical Committee has standardized many extensions over the last decade: from `UNMAP/TRIM` (2011) [16], `ATOMIC_WRITE` (2013) [17], `STREAM` (2017) [18], to a recent one, the NVMe I/O Determinism (**IOD**) interface (2019) [19]. One IOD feature is the *predictable latency mode* (**PLM**) interface, which suggests SSDs work in two alternating modes across time: the deterministic (predictable) and non-deterministic ("busy" for short). IOD-PLM tries to deliver the best I/O latency during the predictable mode and only schedules background activities in the busy mode. For example, a simple use case [14, 15] is to redirect I/Os from a "busy" SSD to NVRAM. The specification does not provide the exact definition of "deterministic window," but a common understanding suggests that in a deterministic window, the device should **not** perform internal activities that would cause unpredictable latencies to user I/Os (*i.e.*, background operations should only be done in the busy window). IOD-PLM is a major leap towards a more open host-SSD collaboration in attacking the latency consistency challenge. Major storage companies and cloud providers are considering the use of IOD [14, 15]. However, it is still considered a "young" interface. Challenges remain on how the host OS and SSDs should be co-designed around this interface.

IOD-PLM is expected to be useful for flash arrays or clusters where the host or applications can redirect I/Os to devices in the deterministic mode, whenever possible. Let's take the read operation on a RAID-5 flash array as an example. Here, an "**unpredictable I/O**" destined to a busy device can be **reconstructed** using the parity and the rest of the data blocks in the same stripe. The reconstruction is done by the "**array's host**" (*e.g.*, software/hardware array controller). Suppose a stripe consists of 3 data chunks (B_0, B_1, B_2) and 1 parity chunk (P), if reading B_0 is unpredictable because device #0 is busy, B_0 can be reconstructed by reading the parity and other chunks within the stripe (B_0=B_1⊕B_2⊕P), with the hope that other devices (#1 to #3) are in the deterministic state. This proactive reconstruction scheme is often referred to as "degraded reads" [10, 20–23], a popular concept used when parity computation is much faster than waiting.

While degraded-reads seem to be straightforward and a natural fit for IOD-PLM, we discovered a number of shortcomings (detailed later) during our journey to exploit the interface for an always-deterministic flash array design.

In this paper, we first pinpoint four limitations (and opportunities) to improve the IOD-PLM interface: (a) PLM is currently treated as a "best-effort" contract (the SSD can autonomously switch from deterministic to busy mode whenever it needs to); (b) the specification states that this interface can return much possible information of the device's PLM status but not many standards available on how the host should use them; (c) PLM is currently configured at a coarse-grained level (whole device or partition) not optimum for modern devices with high channel-level parallelism where some channels might be free from GC momentarily; (d) while PLM allows the host OS to "softly" control how long the device should be in (non)deterministic windows, there is no guideline on how long the windows should properly last.

To this end, we introduce **IODA**[2], an I/O deterministic flash array built on top of small but powerful extensions to the IOD-PLM interface.

IODA introduces three main techniques to enhance the IOD interface and facilitate a deterministic host/SSD co-design incorporating degraded-reads seamlessly: **(1)** predictable mode I/Os for augmenting coarse-grained whole-device level predictability with per-I/O level predictability query via a simple flag ("Will this I/O be predictable? Yes/No"); This allows a more live response of the predictability status to signal the host decisively on whether and when to trigger reconstructions. **(2)** piggybacking busy remaining time for assisting the host in picking less-busy devices for reconstructions in the case of concurrent internal operations. Thus, we only need to wait for the least busy devices to achieve improved latencies; and **(3)** a stronger (un)predictable-latency-window formulation and scheduling scheme for programming a proper upper bound value of the (un)predictable window in every device of the array to guarantee a stronger predictability contract. We show how the combination of these approaches is more powerful than each of the individual methods. Our techniques add only 5 new fields to the existing IOD-PLM interface and NVMe commands (18 lines in the Linux NVMe driver), keep the flash firmware simple (only 60 and 186 lines of new logic on 2 popular SSD platforms [24, 25], respectively), and isolate all the complexity in the host OS, with 1814 new lines in the Linux RAID ("md") sub-system.

We performed a thorough evaluation (§6) with 9 datacenter I/O traces, 6 file system, and 15 popular data-intensive workloads. Compared to the baseline, IODA reduces I/O latency by 1–75× between p95–p99.99 (*i.e.*, the 95–99.99th percentiles) and 1.7–16.3× on average. Compared to an "ideal" scenario where there are no write-triggered GCs, IODA is only 1.0–3.3× slower between p95–p99.99 while the baseline suffers from 1.1–88.3× degradation. To compare IODA with state-of-the-art approaches, we also re-implement 7 published methods that represent preemption [26–28], program/erase suspension [29–31], speculation [32, 33], GC coordination [34, 35], partitioning [36–38], "tiny-tail" controller design [10], and SLO-aware prediction [39].

Overall, our measurements show that IODA provides a strong IOD guarantee (no I/Os delayed by GCs), even under the maximum write burst, and without sacrificing throughput; to the best of our knowledge the *first* flash array design that has achieved so.

For the rest of the paper, we assume flash arrays with some level of redundancy. We use N_{ssd} and k to represent the number of devices and parities (*e.g.*, N_{ssd}=4 and k=1 in a 4-drive RAID-5 array).

2 LATENCY UNPREDICTABILITY IN FLASH ARRAYS

SSD firmware must perform background management operations such as GC, which will cause channel/chip-level read/write contention with foreground (user) read I/Os. To show the GC impact to latency predictability in flash arrays, imagine a typical sequential large read to block addresses B_1 to B_4 that are striped across multiple SSDs. If one of them is "unpredictable" (*e.g.*, must wait for a background operation to finish), then the entire large read will be delayed. Figure 1a and 1c show the cascading impact of a busy SSD (doing GC) to large user I/Os (stripe-IOs) in consumer- and enterprise- level flash arrays, respectively.

Here, we form a RAID-0 on 4 real SSDs (see Figure 1 captions) with 4KB chunk size; we run 16KB full-stripe random reads (foreground). To trigger different intensities of GC (background) noises, we also inject random-write noises of 1, 10, 40, 100, and 800 KWPS (kilo-writes-per-second) where "1W" implies a 4KB random write. Every

[2] IODA is pronounced "**Yoda**," a wise and determined Jedi Master

i^{th} full-stripe read generates 4 page **sub-IOs**. We instrument Linux Software RAID to measure the latency of every sub-IO. Thus, for every i^{th} read, we measure 4 sub-IO latencies $L_{i1}..L_{i4}$ from the 4 SSDs. We then measure the longest delayed (latest) sub-IO with the following slowdown metric: $S_i = Max(L_{ij})/Median(L_{ij})$ where j=1..4. With 4 drives we use the 2nd earliest time as the median. Put simply, S_i represents the *slowdown to wait* for the latest returned page (the tail) in every full-stripe read i.

Let us look at the results from the consumer-level SSD array first. Figure 1a plots the CDF of all S_i on the array of four SamSung SM951s [40], showing that due to background activities, *the latest sub-IO of a full-stripe I/O can arrive multiple times slower than the earlier ones*. The slowdown becomes worse when GC happens more often (1KWPS green vs. 10KWPS blue lines). *e.g.*, with 10KWPS, a sub-IO read is *15× slower* than the median at p97. Under 40KWPS, we see 40× slowdown at p98.

We emphasize that this slowdown is due to GC and not the random user writes (*i.e.*, queueing delays). This is verified by the five (overlapping) thin gray lines marked "**NoGC**" where we convert the user write to a read noise. The gray lines mostly hovering around `x=1` essentially show that the foreground full-stripe reads observe no (1×) slowdown of sub-IO completions. For a fairer experiment, as NAND read latency is around 20× faster than write latency, we also set the read noises to be 20× more intense or 20× larger in size and obtain similar results.

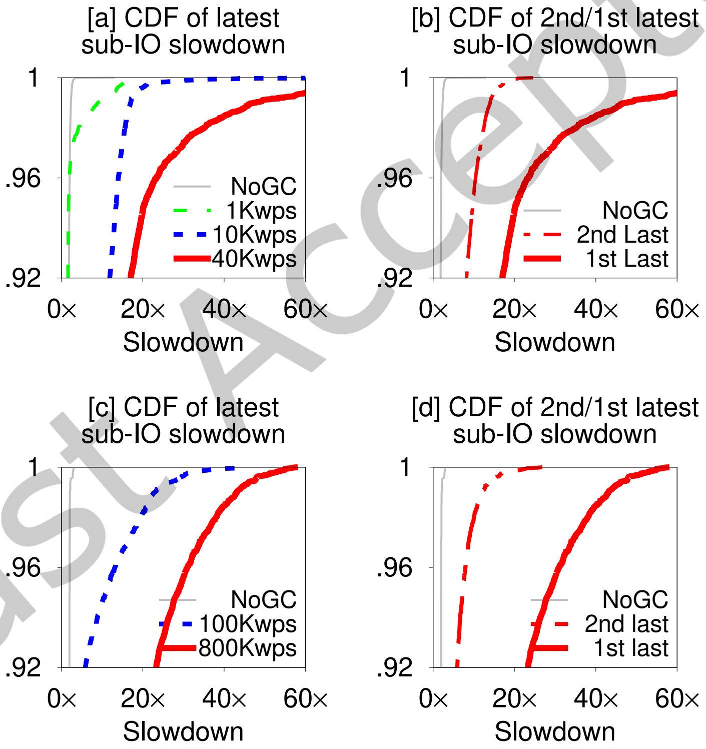


Fig. 1. **GC impact in consumer and enterprise flash arrays (§1).** *The figure shows the CDFs of sub-IO slowdowns on a 4-drive RAID-0. The top figures (**a&b**) is from an array of* consumer *SSDs (SamSung SM951s) and the bottom figures (**c&d**) are from a similar array of* enterprise *SSDs (e.g., Intel P4500).*

Figure 1c is the same experiment as Figure 1a but done on a RAID-0 on top of four enterprise SSDs. Figure 1c shows a behavior on enterprise flash array (as in Figure 1a), but here due to the larger capacity and potentially more advanced techniques employed (*e.g.*, P/E suspension), the GC impacts are observable under a relatively-high intensity write noise at 100KWPS. The aggregate array write bandwidth is 1.2MWPS. Under a more intense write noise of 800KWPS, we can observe worse slowdowns, reaching 60× compared to the normal-case no-GC latencies.

2.1 Opportunity

With the same experiment above, we find a big opportunity to cut unpredictable latency. To show this, we also record the slowdown of the *2nd latest* returned page: $S_i^2 = 2ndMax(L_{ij})/Median(L_{ij})$. Figure 1b and 1d compare the distribution of the 1st- and 2nd-latest slowdowns in consumer- and enterprise-level flash arrays, respectively. For readability, we only show the experiment results with 40KWPS and 800KWPS noises.

As shown in Figure 1b, *the probability that two sub-IOs of a stripe read are simultaneously delayed by GC is much lower than only one page being blocked*. For example, >2× slowdown of the latest page happens *13%* of the time (`x=2` at `p87`), but the 2nd-latest page is >2× slower *only 2%* of the time (`x=2` at `p98`). Thus, if we put this finding in the context of RAID-4/5, *11%* of the slow I/Os can be made fast by reconstructing the late sub-IOs from another SSD that holds the parity block of the stripe. Similarly, the above findings hold true for the enterprise flash array (Figure 1d).

3 IOD-PLM: THE GOOD AND THE BETTER

3.1 How IOD-PLM Works

The NVMe I/O Determinism (IOD) concept [19] introduces two interfaces: "NVM Set" (for isolation, not our focus) and predictable latency mode (**PLM**). PLM suggests SSDs work in two alternating modes across time: **deterministic (predictable)** and **non-deterministic ("busy") windows**. A common understanding suggests that background operations should only be done in the busy window. In more detail, PLM exposes two NVMe commands. First, the "`GetPLMLogPage`" command ("`PLM-Query`" for short) allows the **host OS** (*e.g.*, Linux RAID) to query the device state such as the #I/Os in the future that the device can guarantee to be deterministic in latency. Second, the "`PLM-Config`" command allows the host to toggle the device's deterministic/busy state. *However*, one caveat is that this IOD interface is seen as a "best-effort, soft contract," *i.e.*, the device can autonomously transit to the busy state under certain conditions (*e.g.*, performing GCs when running out of over-provisioning space), hence breaking the predictability guarantee.

3.2 Opportunities for Improvement

PLM is a major leap towards more open host-SSD communication and the interface keeps evolving. We argue it requires further enhancement to enable a principled co-design for strong predictability due to the following deficiencies.

First, `PLM-Query` returns significant information of the device PLM state [19, §8.18] without (so far) much guidance on how the host should use them. Both the host and the device must keep track of this "soft contract," (*e.g.*, extensive inflight I/O status) which can create much management complexities. Let us imagine a return value of "the next R reads and W writes will be predictable." The host must keep track of this information; for example, as long as future writes are fewer than W, the host can still submit many reads (no write-triggering GCs). Likewise, on the SSD side, the firmware logic must be modified to keep the promise by tracking the internal inflight I/O status.

Second, the whole-device non-deterministic mode is unnecessarily too coarse-grained. Modern SSDs have many parallel channels (*e.g.*, 16 or more) where a GC activity on certain channels will not disturb user requests on other channels. However, because the device declares to be busy as a whole, the host might unnecessarily reconstruct I/Os from other devices while the I/Os could have been destined to non-busy channels inside the currently busy device. This limitation would adversely increase overall system resource utilization and jeopardize performance predictability.

	IODA	Preemption	Partitioning	Speculation	Suspension	Coordination	TTFLASH	Prediction
Determinism	✓	✗	✓	✗	✗	✗	✓	✗
Throughput	✓	✗	✗	✗	✓	✓	✗	✓
Transparency	✓	✓	✓	✓	✓	✓	✓	✗
Deployment	✓	✓	✓	✓	✓	✓	✗	✓

Table 1. **Comparison of IODA to state-of-the-art approaches.** IODA *achieves performance determinism without sacrificing throughput, and is transparent to applications with minimal device-side changes for easy deployment.*

Third, the PLM busy window duration is vital for a strong predictability guarantee (more in §4.3), however, we are not aware of any work that attempts to analyze and formulize the proper window size. In particular, we need a "configurable" framework to lay out how these values are derived and program them properly. The PLM's "soft" control of the busy/predictable window transition is far from being ideal.

3.3 Related Work and Our Contributions

Table 1 summarizes existing approaches that attack the flash performance challenges. The popular methods include preemptions [26, 29, 41], hints [39, 42–45], partitioning [11, 37, 38, 46], speculation [22, 47, 48], latency prediction [39, 49], and coordinated GCs [10, 34–36, 50, 51]. Traditional preemptions cannot indefinitely avoid/postpone GCs, as they will revert to normal blocking behavior under insufficient over-provisioning space. Hint-schemes such as [45] require code changes, breaking application transparency. Partitioning methods like FlashBlox [37] exploit parallel hardware resources (channels/chips) to achieve strong isolation at the cost of the aggregate bandwidth drop. I/O speculation techniques, *e.g.*, request cloning or hedging [3] pose the question of *how long* to wait before forcing an I/O reconstruction/replication; it remains challenging to adapt the speculation eagerness for balanced resource utilization and effectiveness. Latency prediction approaches such as MittOS [39] or LinnOS [49] answer the *when*-to-reconstruct question but suffer from inaccuracies without collaboration with the device. Coordinated GCs, as in TTFLASH [10], overcome latency prediction limitations, but introduce another question of *when* every device must start/stop GCs.

In terms of *which layer* tames the SSD performance issues, vast research has been done, from device-only modifications [10, 26, 44, 52–54], host-level changes [32, 35, 36, 55, 56], transparent approaches on programmable devices [12, 25, 39, 57, 58], to interface solutions [18, 51, 59–61]. Device-level proposals usually require vendors to significantly modify the firmware policies, thus not attractive for quick deployment; host-only optimizations can only guarantee a soft contract (*i.e.*, not eliminating background interferences); transparent approaches do not work for commodity SSDs, and many interface-level solutions focus on various types of inefficiencies of the existing software/hardware stack. Fortunately, IOD-PLM interface has been accepted and time is ripe for us to build solutions on it. IODA builds on top of the standard NVMe IOD-PLM interface and only requires minimal firmware changes for easy deployment.

The emerging Zoned Namespace (ZNS) [60] interface offers new opportunities for predictable performance by delegating more device controls to the host, but it could still potentially benefit from IODA techniques to co-schedule housecleaning tasks (*e.g.*, GCs) and the hardware across devices. We leave more detailed study as future work.

TTFLASH [10] tackles a similar problem as IODA. However, IODA's design context, principles, and technical challenges are fundamentally different. TTFLASH is a device-level design while IODA focuses on host/device co-design with minimal interface changes (we must address host-level and minor device-level changes and the interface design). TTFLASH requires extensive controller/firmware re-architecting, which we argue is not realistic (*e.g.*, reliance on NAND "copybacks" to enable chip-level blocking GC but skipping ECC checking that vendors do not employ; GC has to move data from NAND to RAM for ECC checking by the controller). IODA does not enforce a specific GC policy. More importantly, IODA tackles a new problem of PLM management on IOD devices, we must address PLM limitations, design and build the needed software support in the host/OS.

Although several works on IOD begin to appear [14, 62], they mainly target hardware-level partitioning for better workload isolation, none of them address IOD-PLM challenges. We leave more detailed comparisons between IODA and related work in §6.2, qualitatively and quantitatively.

While these existing works without a doubt guide us to our ultimate solution, (*e.g.*, we integrate IODA with degraded reads), to the best of our knowledge, none of the works above answer the following questions: How can we extend and manage the existing IOD features and design proper software support to achieve always-predictable latencies? How should the host and the device agree on a proper PLM window to achieve an optimal result? How should the popular concepts of degraded reads and coordinated GCs be redesigned for future IOD-capable drives? We believe these questions are similar to those around the highly popular concept of tail tolerance/speculative execution [63] that has been extended, re-architected, and re-evaluated for many scenarios [64–68] (far too many to cite here). In the same way, our unique contributions lie in answering the above questions.

4 IODA

We present IODA, an I/O deterministic flash array that is built on top of small and simple extensions around the existing IOD-PLM interface. This section describes our journey one step at a time towards reaching a highly deterministic latency and Section 4.4 puts all the pieces together.

4.1 Design Principles

When designing IODA, we adhere to the following goals and principles:

(a) *Make best-effort predictability stronger to guaranteed predictability.* The IOD-PLM concept is ideal for flash arrays if designed properly; the SSDs in the array can guarantee alternating internal activities and the host can leverage data redundancy for I/O reconstruction such that there is *no* single I/O that will be delayed by GC operations.

(b) *Continue reducing the host-SSD semantic gap.* For stronger predictability, we advocate SSDs to be "array-aware" with more but simple co-design/coordination between the OS and the devices without forcing the device to expose much of its internal proprietary information.

(c) *Make predictability more fine-grained.* To achieve a more efficient array, coarse-grained predictability mode (at the whole device level) should be augmented with finer-grained predictability at the I/O level to alleviate unnecessary reconstruction/rerouting overhead.

(d) *Limit device-level modifications and keep most of the complexity in the host.* Deployed flash firmware has gone through years of development, hence should not be heavily re-architected. All needed is for the firmware to shift its internal activities over time (*e.g.*, <100 lines of change). Similarly, applications should not be modified, leaving the OS to handle all the complexity of guaranteeing strong predictability.

4.2 PL$_{IO}$: Predictable-Latency Flagged I/Os

Our first method is to introduce **PL$_{IO}$**, *predictable-latency flagged I/Os*, by piggybacking a *binary* PLM query within the I/O submission command ("Will this I/O be predictable? Yes/No"). This allows a more live response of the predictability status. In other words, to have deterministic latency, the host ideally should know which I/Os that will be delayed internally by the device such that the host will perform a degraded read without waiting. PL$_{IO}$ binary response serves as a timely and accurate signal for the host to initiate proactive reconstruction. PL$_{IO}$ modifications to the (a) interface, (b) firmware, and (c) host is minimal:

(a) At the NVMe interface level, we extend the I/O submission command with a 2-bit `PL` **flag** (using a slot in the existing 64 reserved bits). The purpose of this bit is as follows. For every user I/O, the host can mark them with `PL=true (01)` hinting to the underlying device that "ideally" this I/O should exhibit a predictable latency (not queued behind GC activities). If predictability cannot be guaranteed, please acknowledge the host as soon as possible. In our flash array setup, we initially set all read I/Os with `PL=true`.

(b) On the device side, when a user I/O contends with GC, the device firmware should *quickly* "fail" this unpredictable I/O by placing `PL=fail (11)` in the corresponding completion command. Afterward, the host can proactively reconstruct this unavailable block from the other devices in the array (§3). Otherwise, if GC is not active, the device can serve and complete the I/O without changing the flag (*i.e.*, the same processing logic as normal I/Os).

(c) On the host side, upon receiving a failed I/O, if the I/O is a read operation, the host can simply reconstruct the unavailable block by submitting additional I/Os with predictability off (`PL=false (00)`), which we call *reconstruction I/Os* to differentiate from the original user I/Os. After reconstruction, the host can return to the upper layers (*e.g.*, file systems) and deem it completed.

4.2.1 Benefits and Limitations. This simple extension delivers a large benefit for two reasons. First, failing an I/O only takes 1μs through PCIe and the `xor`-based reconstruction takes less than 10μs on modern CPUs. Thus, this fast response (plus reconstruction) can provide a significantly faster response than waiting for background operations to complete. The `PL` flag serves as a proactive signal to coordinate the device and the host on the correct timing to respond to the non-determinism. Second, the probability that more than one **sub-IOs** of the same stripe are delayed by simultaneous GCs on different devices is significantly lower than the probability of just one sub-IO getting delayed. A sub-IO is a page I/O within a full-stripe I/O. In a 4-drive array, a full-stripe I/O has 4 sub-IOs, including the parity page. We observed this probability in a detailed profiling experiment in the Linux block layer with real SSDs.

A limitation of this approach is that it can only reconstruct k sub-IOs within a stripe where k is the number of parity blocks (*e.g.*, k=1 in RAID-5 and k=2 in RAID-6). Thus, it is still tail-prone when $>k$ sub-IOs are not predictable (*i.e.*, the reconstruction I/Os also cannot be served quickly). The subsequent sections will address this limitation and show how PL$_{IO}$ can be more powerful under further enhancement.

4.2.2 A Further Extension (PL$_{BRT}$). To address the limitation of PL$_{IO}$, we explored extending the firmware furthermore to return the **"busy remaining time" (BRT)** to inform the host how long the corresponding I/O would have to wait. Thus, when multiple, n sub-IOs are returned with unpredictable flags (`PL=11`), including the reconstruction I/Os, the host will resubmit $n-1$ of the sub-I/Os with the *shortest busy remaining time*. This time, these I/Os must be resubmitted with `PL=00` to avoid recursive fast-failures (*i.e.*, these I/Os will wait for GCs if any). In the firmware, calculating the BRT that affects a particular incoming I/O can be done in a straightforward fashion because it is about the chip and channel-level queueing delays with established device-level specifications. In the NVMe interface, we piggyback the busy remaining time in the NVMe completion command of the affected I/Os (using the 64 reserved bits).

Later in the evaluation, we show that PL$_{BRT}$ improves upon PL$_{IO}$, but PL$_{BRT}$ fails to provide a strong predictability contract. The PL$_{BRT}$ technique works effectively under a low probability of multiple I/Os in a stripe delayed by concurrent background operations. However, we observed that in some deployments of a major storage company, the flash array design absorbs user writes to a separate battery-backed DRAM and flushes them in large sequential full-stripe writes across the SSDs. Hence, all the SSDs in the array *age at the same pace*, and because the device models are usually the same (*e.g.*, same firmware logic), GC operations kick in at relatively the same time. PL$_{BRT}$ becomes ineffective here because the host would see multiple unpredictable I/Os with *similar* busy remaining time values.

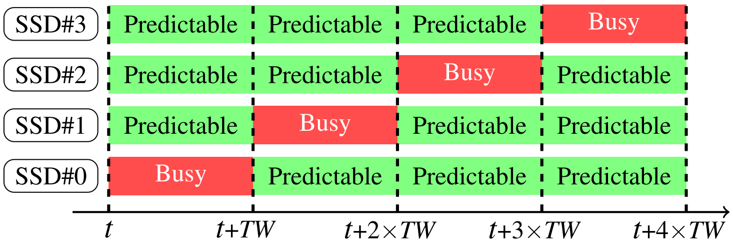


Fig. 2. **Alternating busy/predictable windows (§4.3).** *This figure, using a 4-drive RAID-5, shows that in any time window (a duration of TW), there is at most one device in the busy mode, performing GCs.*

$$TW \leq \frac{R_p \times S_t}{(N_{ssd} \times Min(B_{pcie}, Max(\frac{N_{dwpd} \times (1 - R_p) \times S_t}{8hours/day}))) - (\frac{(1 - R_v) \times N_{ch} \times S_{pg} \times N_{pg}}{(t_r + t_w + 2 \times t_{cpt}) \times R_v \times N_{pg} + t_e})}$$

Fig. 3. **The TW formulation.** *The formula depends on 11 hardware-level parameters and 3 workload related parameters (the width of the flash array (N_{ssd}), R_v, N_{dwpd}). The breakdown of the formula is detailed in Table 2. SSD controllers can use this formula to calculate and report the TW length (upperbound) to the host during array initialization/creation phase. Later, the host can program appropriate TW value to all the devices in the array.*

4.3 PL$_{Win}$: Busy Latency Windows

4.3.1 Overview. To provide a strong predictability contract, we leverage the fact that the notion of "PLM windows" has been accepted in the NVMe specification, *i.e.*, a device should alternate between busy and predictable windows. We take this concept within the context of flash arrays. Here, we concisely introduce the *rules to achieve strong predictability:*

(1) During the **busy time window** (TW), the device must have time to reclaim enough space via GCs and bring back the free over-provisioning space to a certain level (some percentage of the total raw NAND capacity) to serve the incoming writes during the predictable window.

(2) During the **predictable time window**, which lasts $(N_{ssd} - k) \times TW$ (explained later), every device must have enough over-provisioning space to absorb the largest possible write bursts to the device, hence guaranteeing no GCs are triggered during the predictable window.

Figure 2 illustrates the goal of using TW in a 4-drive RAID-5 array. In the first time window, between time t to t+TW, device #0 enters the busy mode for TW and performs GC to create a large free space in the over-provisioning area, which is crucial for absorbing the maximum write bursts during the predictable window. It's important to note here that the other devices (#1-#3) must be in the predictable mode and may *not* perform any GC. In the next time window, between t+TW and t+(2×TW), device #0 switches to predictable mode while device #1 enters its busy period (taking its turn to do GC operations). In this 4-drive RAID-5, every device must be able to sustain user write bursts within the predictable duration (3×TW) without triggering internal busyness. Note that writes are allowed during both the predictable and busy windows as we do not perform any write throttling or orchestration/staging which limits write throughput. To generalize the TW synchronization across SSDs, given an array's width (N), start-time (t), and TW, the i^{th} SSD will enter its busy state at time $(t + (i - 1 + k \times N) \times TW)$ for k in $[0, 1, 2, ...]$. Each SSD can use the controller's timer to perform busy/predictable state transitions *periodically* (e.g., via timer events) and autonomously without overlapping with other SSDs.

Symbol	Longer Symbol	Unit	Symbol Equation	Sim	OCSSD	FEMU	970	P4600	SN260
Hardware Time Specification									
t_{cpt}	TimeOfChannelPageTransfer	μs		40	60	60	40	60	60
t_w	TimeOfNandPageWrite	μs		2400	1440	140	960	2000	1940
t_r	TimeOfNandPageRead	μs		60	40	40	32	60	50
t_e	TimeOfNandBlockErase	ms		8	3	3	3	6	3
B_{pcie}	BandwidthOfPCIe	GB/s		4	8	4	4	8	8
Hardware Space Specification									
S_{pg}	SizeOfNandPage	KB		16	16	4	16	16	16
N_{pg}	NumberOfPagesPerBlock	.		512	512	256	384	256	256
N_{blk}	NumberOfBlocksPerChip	.		2048	2048	256	2731	5461	4096
N_{chip}	NumberOfChipsPerChannel	.		4	8	8	4	8	8
N_{ch}	NumberOfChannels	.		8	16	8	8	12	16
R_p	RatioOfOverProvisioning	.		0.25	0.12	0.25	0.20	0.40	0.20
R_v	RatioOfGCValidPages	.		0.5	0.75	0.7	0.75	0.75	0.75
Derived Values									
S_{blk}	SizeOfNandBlock	MB	$S_{pg} \times N_{pg}$	8	8	1	6	4	4
S_t	SizeOfTotalNandSpace	GB	$S_{blk} \times N_{blk} \times N_{chip} \times N_{ch}$	512	2048	16	512	2048	2048
S_p	SizeOfProvisionSpace	GB	$R_p \times S_t$	128	246	4	102	819	410
Garbage Collection									
T_{gc}	TimeToGCOneBlock	ms	$(t_r+t_w+2\times t_{cpt})\times R_v\times N_{pg}+t_e$	658	617	57	312	425	408
S_r	SizeOfGCReclaimedSpace	MB	$(1 - R_v) \times S_{blk} \times N_{ch}$	32	32	2	12	12	16
B_{gc}	BandwidthOfGCCleaning	MB/s	$S_r \ / \ T_{gc}$	49	52	35	38	28	39
Workload Behavior									
N_{dwpd}	NumberOfCommonDWPD	.		10	10	40	10	10	10
B_{norm}	BandwidthOfWorkloadWrite	MB/s	$N_{dwpd} \times (S_t$ - $S_p)$ / (8 hours)	137	641	17	146	437	582
B_{burst}	BandwidthOfFullWrite	MB/s	$Min(B_{pcie}, Max(B_{norm}))$	3200	4000	536	3200	3204	4000
RAID									
N_{ssd}	NumberOfSSDsInTheArray	.		8	4	4	8	4	4
Time Window									
TW_{norm}	TimeWindowNormal	ms	$S_p \ / \ (N_{ssd} \times B_{norm} - B_{gc})$	6259	5014	6206	4622	24380	9171
TW_{burst}	TimeWindowBurst	ms	$S_p \ / \ (N_{ssd} \times B_{burst} - B_{gc})$	256	790	97	204	3279	1315

Table 2. **Time window (TW) breakdown and values (§4.3).** *The top row segments are basic NAND/controller-level parameters (i.e., "Hardware Time/Space Specification"), and the bottom row segments (i.e., "Derived Values, Garbage Collection, down to Time Window") are calculated based on the upper rows. We show analysis results for 6 SSD models (the right-most columns).*

While similar coordination ideas as in Figure 2 have appeared in scenarios ranging from in-device RAIN [10, 20, 69] to even distributed "Java GC" [70, 71], we are not aware of existing works that apply it to flash array designs. The unique challenge here lies in programming the proper PLM windows without breaking the predictability contract. In this context, we need a configurable framework to program and formulate the busy window that IOD arrays can base on. For example, an SSD vendor employing a certain GC policy can slightly tune the formula/parameters to achieve the ideal window length for their SSD models; a flash array operator might want to relax the window value to better suit their target workloads for better device lifetime. To this end, we introduce **PL**$_{Win}$, *a TW formulation that flash array's host and devices can use to guarantee the contract*, hence making the flash array deliver predictable latencies all the time.

Due to the complex and proprietary GC dynamics whose details are invisible to the host for modern SSDs, devices are the ideal candidates to calculate the proper TW length and advertise them to the host. Host-based TW calculation would make more sense if devices are willing to expose more of the internals (*e.g.*, ZNS SSDs [60]).

4.3.2 TW Upper-Bound Formulation.

This section describes our TW formulation in a top-down fashion. To satisfy the contract rules, TW must satisfy the following constraint:

$$TW \leq S_p \; / \; ((N_{ssd} \times B_{burst}) - B_{gc})$$

Without losing generality, let's consider a "*full cycle*" of $N_{ssd}{\times}TW$ as illustrated from time t to t+$(4{\times}TW)$ in Figure 2 for one SSD in the array. B_{burst} represents the "*per-device maximum user write burst*", which we will break down in the subsequent section. The SSD is only allowed to perform GC on its own turn (in one TW), while writes can keep coming within the full cycle without any throttling/arbitration until the SSD has a chance again to perform GC. Thus, $N_{ssd}{\times}B_{burst}$ represents the maximum user write burst within a cycle for one SSD. Within its time window, the SSD can run GCs freely to reclaim space, say at the speed of B_{gc} (expanded later). This means $(N_{ssd}{\times}B_{burst}){-}B_{gc}$ is the *net write load* that an SSD should handle in a cycle. In other words, the net incoming write load should not take up all the free over-provisioning space (S_p) that the SSD has.

All combined, the time window length (TW) must be less than the size of the over-provisioning space (S_p) divided by the net write load, hence the constraint above. Given that S_p is typically a fixed size, TW is mainly decided by N_{ssd}, B_{burst} and B_{gc}. For example, under a wide array (large N_{ssd}), TW must be set smaller to avoid breaking the IOD contract (will be analyzed further later).

TW has a lower bound, the latency of the smallest, non-preemptible unit of GC activity (T_{gc}). For example, a firmware might prefer to clean one NAND block as an uninterruptible activity to reclaim enough space within one TW.

4.3.3 TW Parameters.

We now break down our TW formulation in a bottom-up fashion. The parameters we introduced (S_p, B_{burst} and B_{gc}) are high-level parameters that must be derived from hardware specifications. Figure 3 shows our final TW formulation that requires 11 hardware-level parameters. For understandability, we break down this equation in Table 2. The first row segment of Table 2 lists the hardware *time*-related specification such as channel transfer (t_{cpt}), NAND write (t_w), read (t_r), and erase (t_e) time and the host-device PCIe bandwidth (B_{pcie}). The second segment lists the hardware *space*-related specification such as the page size (S_{pg}), pages per block (N_{pg}), blocks per chip (N_{blk}), chips per channel (N_{chip}), number of channels (N_{ch}), over-provisioning ratio (R_p), and the average ratio of valid pages in victim blocks (R_v). These low-level parameters are needed to derive higher-level parameters (the third row segment) such as block size (S_{blk}), total NAND size (S_t), and over-provisioning space (S_p). From here, we can calculate GC behavior (the fourth row segment) such as the time to clean one victim block (T_{gc}), the size of the reclaimed space (S_r), and GC cleaning bandwidth (B_{gc}). The device also needs to understand the workload intensity, such as the maximum write bandwidth (B_{burst}) which depends on two values (B_{pcie} and B_{norm}). More importantly, TW depends on the width of the flash array (N_{ssd}).

4.3.4 TW Example Values.

With all these parameters, we can set TW=TW_{burst} (the last row in Table 2) to fully guarantee the contract. To get a sense of the actual possible values, columns 5–10 of Table 2 show parameters of 6 SSD models we analyzed, including a simulated device that mimics a consumer SSD ("Sim"), a flash emulator used for our firmware prototyping ("FEMU") [24], an OpenChannel-SSD ("OCSSD") [25] whose parameters are publicly known, and 3 commercial SSDs from different vendors. We used an SSD prober [72] to profile the hardware parameters of the commercial SSDs. Some of the SSD internal parameters are known to be "guessable" based on the observed latencies [73]. The average number of valid pages in a victim block (R_v) is estimated from running our workloads (§6) in FEMU [24] and OpenChannel-SSD [25]. We emphasize that the use of these numbers are only for analyzing possible TW values on real devices.

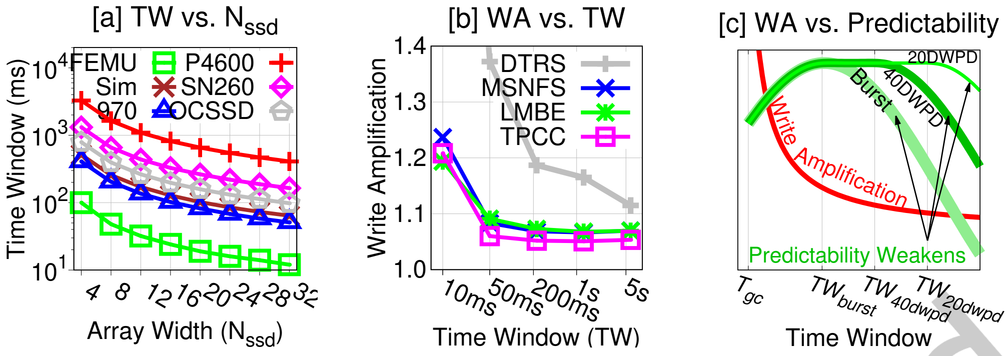


Fig. 4. **Time window analysis.** *Figure (a) shows that TW can scale well to large arrays (e.g., >20 devices); Figure (b) demonstrates improved WA under larger TW; and Figure (c) presents the tradeoffs to balance WA, predictability and TW. Y-axis for the red line represents WA and Y-axis for the green lines denotes latency predictability gaurantees (higher better).*

Overall, when varying the number of devices (N_{ssd}) in the array from 4 to 8, TW_{burst} can range from ~100ms to ~3secs for different SSD models, which gives us a reasonable window length large enough to run sufficient GCs. Higher capacity devices such as enterprise SSDs can have a longer TW, primarily because they have more over-provisioning space to absorb the incoming write burst, but the maximum user write burst (B_{burst}) is also limited by the PCIe bandwidth.

Note that we use fixed parameter values to simplify the analysis without losing generality. For more complex scenarios where some parameters (*e.g.*, R_v) will change over time due to workload and/or GC dynamics, we believe SSD vendors can further tune and customize the TW formula to derive more accurate TW values for their device models. This is because SSD vendors have full control of their firmware/GC policies. For example, the vendors could use a "worst-case" R_v value to calculate the tightest TW upper-bound. Later, our FEMU-based evaluation (§6) shows the TW approach can help us achieve predictable latencies.

4.3.5 TW Scalability & Write Amplification (WA). We now analyze the trade-offs of TW values. Figure 4a shows the implication of larger array width (x-axis) to the TW value (y-axis) of 6 device models in Table 2. A *wider* array (larger N_{ssd}) forces TW to be *lowered* as the predictable window duration ($N_{ssd} \times TW$) for every device increases while the busy window period remains the same ($1 \times TW$). This means the over-provisioning space will be full relatively faster.

Unfortunately, as shown in Figure 4b, a *lower TW* (in x-axis) causes a *higher* write amplification (**WA**) factor (in y-axis). Here we ran various workloads on SSD model "Sim" (more in §6.3.7). Let's take an example of TW=100ms in a 4-drive RAID-5 array, which implies a 300ms of predictable window length for each device. However, user write workload is typically *less intensive* than the maximum possible write burst, thus the over-provisioning space might *not* be full after 300ms, but yet the device is *forced* to transition to the busy window and start cleaning despite not many pages to clean, which then increases WA.

4.3.6 A More Relaxed Contract to Reduce WA. With the above analysis, a flash vendor/operator might worry about the unnecessary high WA given a small TW value. A preferable way is to absorb as many (over)writes until the over-provisioning space is almost full before starting GC. To incorporate this, the flash array can reuse our formulation but replace the maximum write burst (B_{burst}) with a typical "normal" user write throughput (B_{norm}). An industry standard to set this number is by using the drive-write-per-day metric (DWPD) [74]. For calculating B_{norm} in Table 2, we use DWPD values of 10 to 40 (N_{dwpd}), often suggested to prolong the device lifetime to 3-5 years [75].

Plugging in this value to the same formula will give us TW_{norm}. As shown in the two last rows of Table 2, TW_{norm} increases the busy window length by 6-64× (higher compared to TW_{burst}), hence a longer predictable window length. While this reduces write amplification, we must call this a "*relaxed*" (weaker) contract. The reason is that the user write intensity may jump higher than the expected B_{norm} bandwidth. This in turn will fill up the over-provisioning space quickly and force the device to trigger GC even when it's not supposed to (still in predictable mode). This will be a rare event if user workloads follow the suggested DWPD.

4.3.7 The WA and Predictability Tradeoff. As analyzed above, one might prefer to re-configure the TW to achieve low WA without breaking the predictability contract. Figure 4c illustrates the tradeoff between write amplification and predictability under different time window values (x-axis). While WA improves with larger TW (red line), the predictability guarantee weakens if the TW is excessively too large as GCs have to forcefully kick in. Thus, it's necessary to find the sweet TW spot/range that can satisfy both requirements.

Under a "Burst" workload (boldest green line), the predictability guarantee first increases (*i.e.*, delivering overall better tail latencies) starting with the lower-bound TW value of T_{gc}, and peaks around TW=TW_{busrt}, the tight upper-bound TW value under the maximum-possible burst load. As TW continues to increase, the predictability guarantee weakens/decreases. For a lighter load (*e.g.*, the 40 and 20 DWPD green lines), they show a similar predictability-vs-TW trend, but the peak predictability guarantee can sustain over a range of $TW \in [TW_{burst}, TW_{40dwpd}]$, or $[TW_{burst}, TW_{20dwpd}]$, respectively. Here, TW_{40dwpd} represents the calculated TW value based on Figure 3 with N_{dwpd}=40. Combining the red line WA trend, the flash array operators better switch the TW from TW_{burst} to TW_{40dwpd} for better WA if the workload intensity decreases from "Burst" to "40DWPD". To re-configure TW, all needed is an NVMe admin command to re-program the TW value for all the devices in the array), and it can happen at the granularity of time slices (*e.g.*, every few minutes) or per-workload, which flash array operators already have good control of. Furthermore, the OS can be strengthened to dynamically adjust TW based on load changes. However, when under bursts, unpredictability will still show up as the TW adjustment lags behind workload intensity changes.

4.4 Putting It All Together

In summary, we show that the two combinations of PL$_{IO}$ + PL$_{Win}$ creates a very efficient flash array that fulfills the two rules of the strong contract we mentioned in Section 4.3. When not combined, each of these two techniques has limitations (which we will evaluate later).

PL$_{IO}$ only: As discussed before, this method advocates a "fail-if-slow" hardware design to enable host-level timely reconstruction for better latencies. However, it does not prevent multiple sub-IOs within a stripe from concurrent GC delays in different SSDs, thus the inability to achieve predictable latency when multiple SSDs are busy.

PL$_{Win}$ only: Although PL$_{Win}$ by itself guarantees at most one busy SSD in every busy time window, this labeling is *too coarse-grained*, *i.e.*, an I/O destined to a busy SSD might not contend with the internal GC. Let us suppose a block read B_0 to a busy device D_0 that must read the data via channel #8 in D_0. Channel #8 may be idle because the GC activities currently are on the other channels. But because PL$_{Win}$ *assumes the whole* device D_0 is busy (too coarse-grained), then the host will not send B_0 to D_0. As a result, B_0's data must be reconstructed by reading B_1, B_2 and the parity block P. In general, because the host will never send any I/O to an SSD in its busy window, the frequent parity-based reconstruction overhead (probabilistically 25% of the time in a 4-drive array) is unnecessarily too excessive.

IODA (PL$_{IO}$ + PL$_{Win}$): When PL$_{IO}$ and PL$_{Win}$ are combined, the host will always send I/Os with `PL=true (01)` *even to* a device in the busy state. It is more opportunistic in a more fine-grained way—*predictability is per I/O, not the whole device (or partition).* If the I/O going to the busy device is not contending with GCs, then no data reconstruction is necessary. Otherwise, the array will guarantee that every busy I/O (`PL=fail (11)`) can always be circumvented. With IODA design, we ensure that only non-deterministic I/Os contending with GCing channels in the busy SSDs will be fast-failed and reconstructed from other drives. The reconstruction I/Os are guaranteed to be predictable based on our PL$_{Win}$ window formulation, so they will not bloat up the system with endless/nested extra traffic. Later in the evaluation, we show that IODA caps the extra load to only a small percentage (*e.g.*, 6% in Figure 10b, and with <10% fast-rejected reads in Figure 8 across all the workloads). The CPU overhead is negligible compared to GC-induced long I/O latencies. Given this per-I/O predictability, our final IODA design also does *not* degrade the original aggregate bandwidth (IODA bandwidth is close to the raw RAID-5 bandwidth).

Regarding, PL$_{BRT}$ (the shortest-background-remaining-time strategy), as stated in Section 4.2, we no longer need it but will still evaluate it for SSD vendors who do not prefer SSDs to be array-aware. (In PL$_{Win}$, the TW calculation requires the SSDs knowing N_{ssd}, the number of devices).

Interface and control flow: To achieve PL$_{IO}$+PL$_{Win}$, we extend the NVMe IOD-PLM and submission/completion interfaces with only 5 simple fields. Upon array initialization, the host informs each of the devices three pieces of information, array type (*e.g.*, k=1 in RAID-5) and the array width via two new fields, `(1) arrayType` and `(2) arrayWidth`. Next, the device plugs in these values to program TW internally and returns the value via `(3) busyTimeWindow` field in the `PLM-Query`'s response. (Device proprietary information is not exposed to the host). During runtime, the host and the device can tag submission (and completion) commands with the `(4) PL` flag. For flexible array volumes, the host can submit a new `arrayWidth` and the devices can re-program the `busyTimeWindow`). Finally, the host and the devices communicate the `(5) cycle's start time` (t in Figure 2).

Write path: IODA does not change the way the host/array or device performs writes. Data are striped and each write will trigger the parity updates. For non-full-stripe writes, parity updates will trigger RAID-level read-modify-writes. In this case, the reads are tagged with the `PL` flag. Since writes usually tend not to be latency sensitive, IODA design mainly targets strong read performance predictability without degrading the array's aggregate write bandwidth. IODA does not rely on write staging/orchestration. The TW analysis in §4.3 holds true for the general case where writes can arrive at the devices in both predictable and busy windows freely. IODA also does not change the write semantic/crash-consistency of the array. For example, if Non-Volatile Memory (NVM) (*e.g.*, NVRAM [76] or Optane Memory [77]) is used, the host/array only needs to write data to the NVM and flush to the device later. Otherwise, writes are directly acknowledged either when hitting the in-device buffer or the NAND pages when device buffers are full. We acknowledge that NVM can be used as an effective caching layer and greatly improve average latencies; but the tail latencies which are often caused by cache misses, unfortunately, will not go away. For example, read misses will still contend with GCs (triggered by frequent flushes) at the SSD/array level. This is because write buffering (*e.g.*, using NVRAM) only removes user-level read vs. user-level write contention. With/without write buffering, user-level reads are still contending with GC-induced writes (which is our focus). Our current IODA prototype is built on top of the Linux "`md`" sub-system without NVRAM support.

Limitations and discussions: We assume the SSDs in the array are of the same model and size. The SSD vendors should be persuaded to implement our simple interface extension. IODA does not cut tail latency due to I/O bursts (*i.e.*, host-side queueing delays); it only removes non-deterministic latency due to GC activities (which is a major goal of IOD-PLM). While IODA currently concentrates on GC-induced non-determinism, it can be extended to handle other types of I/O contentions (*e.g.*, queueing delay, wear-leveling, flushing, etc.), apply to other types of array layout (*e.g.*, erasure-coded systems for more flexible busy window scheduling), and benefit new hardware determinism-capable designs via PL$_{IO}$ (*e.g.*, head-of-line blocking in networking).

5 IMPLEMENTATION

We now describe IODA implementation [78].

IODA's firmware side: We prototype the firmware logic in two open-source SSD research platforms. **(a)** *FEMU (upgraded).* FEMU is a recent QEMU-based and DRAM-backed SSD emulator [24, 79] used by some recent works appearing in top venues [44, 45, 80, 81]. To make FEMU resemble modern SSDs, we had to make several optimizations in *1200 LOC*. First, FEMU's default FTL imposes a high computational overhead which causes inaccurate emulation under high user load. Thus, we **(1)** implemented a new page-level FTL optimized for FEMU emulation model, **(2)** offloaded the FTL logic to a separate polling thread to avoid interference from other management logics, and **(3)** re-implemented the data placement and GC policies taken from modern SSD designs [25, 82]. Second, we had to extend the firmware emulation with more basic features such as write buffering and flushing policies (*e.g.*, LRU with a balanced binary search tree) and preemptive GC policy. All of the upgrades and extensions now allow us to build IODA's firmware logic in FEMU as well as to rapidly re-implement other related works for evaluation purposes. Bottom line, our FEMU version can deliver 400 KIOPS throughput and as low as 10μs I/O latency. On top of this, we then built a firmware that returns the `PL` flag and performs GC only in the busy window, all only in *60 LOC*. FEMU allows fast prototyping but one drawback is its DRAM-backed emulation nature (*i.e.*, not a real SSD). Thus, we also implemented IODA on real hardware platform to validate our designs. **(b)** *LightNVM+OCSSD.* We prototype IODA with LightNVM on a real OCSSD [25, 83] in 186 LOC for additional evaluation. One design flaw of our OCSSD controller is that it excessively favors reads over writes (*e.g.*, write throughput drops to only 3MB/s under a 2:1 read/write mixed workload). To address this issue, we re-architected LightNVM with a per-chip FIFO queue in 780 LOC.

We also explored other popular hardware platforms, such as OpenSSD [84] and DFC [85], but found that they are not appropriate platforms to implement IODA.

OpenSSD: The most ideal platform to implement IODA is the OpenSSD [84] platform where we can modify the FTL logic and the NVMe interface (a blackbox design like exisiting commercial SSDs). However, OpenSSD's programming framework is a *single-threaded* C implementation of the controller, which on the positive side speeds up FTL research development, but on the negative side not enabling more complex implementations. For example, in OpenSSD, when the SSD is doing GC, the controller cannot be programmed to concurrently read the submission queue and return a busy signal. This simple programming model could not handle concurrent operations. We tried many approaches to work around this, but at the end discard using OpenSSD.

DFC Card: Dragon Fire Card [85] is another SoC-based platform where the firmware changes can be implemented in the "mini" Linux on running on the SoC. Earlier DFC cards can directly manage NAND chips (the on-SoC Linux has an FTL driver), but it's no longer supported; The latest version of DFC cards directly attach to off-the-shelf SSDs where the FTL now resides in the SSD firmware, invisible to the user side. DFC cards lately are used as research platform to show near-storage processing, rather than pure FTL research.

IODA's host side: The host-side logic is written in *1814 LOC* in Linux 4.15 Software RAID (*i.e.*, the `md` subsystem) and *18 LOC* in the NVMe driver. While the LOC is small, it took us a long time to address many hurdles in the complex Linux storage stack such as the intricate timeout/retry mechanism, the `NVMe/BIO/request` I/O `PL`-flag passing, and the complex per-stripe state machine.

Re-implementation of other works: It is important to compare IODA comprehensively, but because other works use varying platforms (some even cannot run), "apples-to-apples" comparison would be difficult to make. With our upgraded FEMU, we were able to re-implement state-of-the-art techniques [26, 30, 32, 34, 36] in around *3400 LOC*. Here we provide more details on how we implement related-work on FEMU stack (the changes are either in the Linux kernel or FEMU).

- **Proactive/Cloning [3, 32]:** Despite user I/O size, proactive methods [32] issue full-stripe reads to the array. It only waits for the first few returned I/Os which are enough to reconstruct user data before marking the user I/O done and returning to upper layers. This simplies the overall design and is potentially more deterministic to deliver fast I/O latencies, but at the cost of extra I/Os which might overload the devices. We implement the proactive mechanism in the Linux Software RAID layer, with only kernel-level changes required.

- **Harmonia [34]:** Harmonia [34] utilizes a global GC policy to coordinate GCs in different SSDs to start at the same time. Harmonia helps lower the overall possibility that I/Os will be blocked by GCs, thus improving performance. We implement Harmonia in FEMU with a global GC control logic. Whenever it detects GC is triggered in one SSD, it will simultaneously trigger GCs on other SSDs in the array. Harmonia only requires FEMU-level modifications.
- **Preemptve GC [26] and P/E Suspension [29]:** With preemptive GC (`PGC`) [26], user reads can be interleaved with GC reads, writes and erases to alleviate GC inteference to user reads. It pauses write/erase operations temporarily to prioritize reads, and then resumes write/erase execution. However, when running out of free space, both `PGC` and suspension become ineffective as they have to be frequently kicked in to reclaim more space and block user request. We implement `PGC` and P/E suspension in FEMU by enhancing FEMU's timing model for more fine-grained timing emulation and adding low-level queue structure and asynchronous event support for `PGC` requests.
- **Flash on Rails [36]:** Flash on Rails ("Rails" for short) [36] devides SSDs in an array to read-only and write-only modes and switch their roles periodically (*e.g.*, every 5s). It relies on a write buffer to stage incoming writes before they can be safely flushed to the write-mode drives. We implement Rails using two emulated FEMU SSD instances, with a host-level write buffer sitting in front of FEMU FTL logics. While writes will be directed to the write buffer, reads will be directly sent down to the read-only FEMU drive, with no interference from writes.

6 EVALUATION

We present our comprehensive evaluation in three sections: We first show the main results of latency determinism brought by IODA approaches under various workloads (§6.1). Then we present comparisons of IODA with the state of the art (§6.2) and show extended evaluations (§6.3).

Platform setup: Most experiments are done on FEMU (for reasons mentioned in §5) running on Emulab D430 machines [86]. We run Linux Software RAID-5 (4KB chunk size) on 4 FEMU drives. The LightNVM+OCSSD full-stack setup is similar and done on our local lab machine.

The FEMU's base firmware uses a page-level dynamic mapping and a greedy-GC policy for best cleaning efficiency. GCs are triggered upon reaching a pre-configured high watermark (25% of free blocks available). GCs will forcefully run at full speed under the low watermark (5%) to ensure enough free space for user requests, *i.e.*, user request processing will be stalled until the number of free blocks resumes to the high watermark level. The device parameters were detailed in the "FEMU" column in Table 2. We configured FEMU to emulate modern low-latency SSDs (*e.g.*, Z-NAND [87]) with SLC-like access latencies (*i.e.*, ~200μs for writes), faster than existing MLC/TLC SSDs analyzed in Table 2. Later, we show that IODA evaluations on our MLC-based OCSSD show the same conclusion as FEMU.

Macrobenchmarks: For block I/O traces, we use 4 SSD traces from Microsoft data centers, spanning cloud storage (`AZURE` and `COSMOS`), search engine (`BingIdx`) and database workloads (`BingSel`) and 5 standard SNIA block traces [88] that we have re-rated 8-32 times more intense to reflect modern SSD workloads, all characterized in Table 3. In these traces, we pick the 1-hour busiest period. For real applications, we run 6 Filebench workloads [89] and 3 YCSB/RocksDB workloads [90] on the ext4 file system. Additionally, we also run 12 other storage workloads ranging from GNU applications, Sysbench [91], to MapReduce (Hadoop/Spark) workloads [92]. All user I/Os are marked as latency sensitive (`PL=true (01)`).

Metrics: We primarily report read latencies for the block traces and application-specific metrics for the rest (*e.g.*, average latencies, runtime, etc.). We also analyze other aspects of IODA design (*e.g.*, write latency and throughput). Each experiment is repeated and ran for a long period with thousands of GCs triggered over FEMU drives in steady state, showing consistent results. Finally, **pYY** implies the YY^{th} *percentile*.

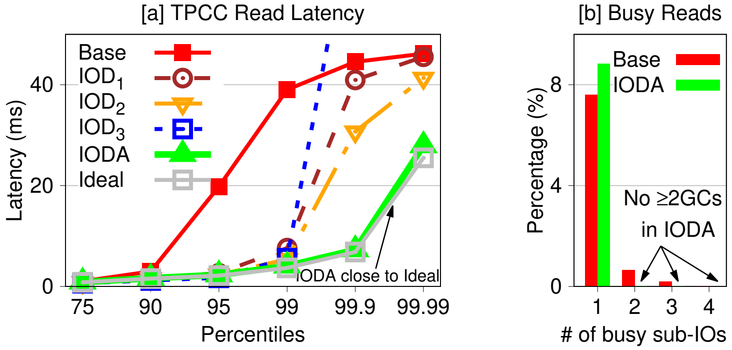


Fig. 5. **IODA percentile latencies and #busy sub-IOs with TPCC (§6.1.1).** *Figure (a): read latencies (y-axis) at major percentiles p75 to p99.99 (x-axis) with various* IODA *strategies. Figure (b): the percentage of stripe-level reads (y-axis) that experience 1 to 4 busy sub-IOs (x-axis).*

6.1 Main Results

This section shows the improvement made by the combination of IODA strategies, one at a time: "IOD$_1$" represents only predictable-latency flagged I/Os (PL$_{IO}$ in §4.2), "IOD$_2$" the shortest-busy-remaining-time strategy (PL$_{BRT}$ in §4.2), "IOD$_3$" the alternating busy windows *only* (PL$_{Win}$ in §4.3, *without* PL$_{IO}$/PL$_{BRT}$), and "IODA" the final approach (PL$_{IO}$+PL$_{Win}$ as described in §4.4). For IOD$_3$ and IODA, our FEMU-based firmware uses a busy time window of 100ms as calculated in Table 2. We also show "**Ideal**" to represent an ideal performance where there are no GC-induced latencies, by disabling GC delay emulation in FEMU.

6.1.1 IODA *Techniques, 1 Workload First.*

For figure simplicity, we first show only the results of using one workload, TPCC (Table 3). Figure 5a shows the latencies at major percentile values (p75 to p99.99) of five different approaches: **(0)** The red Base line represents the TPCC workload without any strategies. Starting at p95 (x=95) the Base's latency is no longer deterministic, consistent with what we observe on real commodity SSDs. **(1)** The brown IOD$_1$ line shows that by just circumventing the busiest (slowest) read, via proactive data reconstruction as signaled by the PL$_{IO}$ method, the latency is more predictable up to p99. **(2)** The orange IOD$_2$ line shows that the PL$_{BRT}$ busy-remaining-time approach further helps but cannot completely evade concurrent busyness. **(3)** The blue IOD$_3$ line shows that PL$_{Win}$-only method is stable up to p99 but it is expensive (spikes at p99.9 and higher) due to the excessive and unnecessary data reconstruction (§4.4). **(4)** **[Key result #1]** *Finally, the bold green* IODA *line in Figure 5a shows that PL$_{IO}$+PL$_{Win}$ provides the best latencies. The thin gap between the* Ideal *and* IODA *lines shows the power of* IODA *in being latency deterministic. Even at* ***p99.99****,* IODA *is only* ***9%*** *slower than the ideal performance.*

Figure 5b reveals the reason behind IODA's success. The x-axis shows how many "sub-IOs" of a stripe are returned busy (PL=11, §4.2). At x=1, the Base bar shows that roughly 7% of stripe-level I/Os experience 1 busy sub-IO, but the base approach just waits for (does not reconstruct) busy sub-IOs. At x=2, Base shows that almost 1% of the stripe-level reads experience 2 busy sub-IOs. While IOD$_1$ and IOD$_2$ can reconstruct 1 busy sub-IO, it cannot evade this concurrent busyness. That is why the IOD$_1$ and IOD$_2$ lines in Figure 5a start increasing between the p99 and p99.9 values. **[Key result #2]** *With our final approach, the green* IODA *bar in Figure 5b shows that our time-window approach successfully* ***shifts*** *concurrent GCs across time such that at any time there is at most only one busy sub-IO per stripe.* Hence, the IODA bar is higher than the Base bar, reaching y=8% at x=1 but y=0 at x>1 (acceptable given the reconstructability).

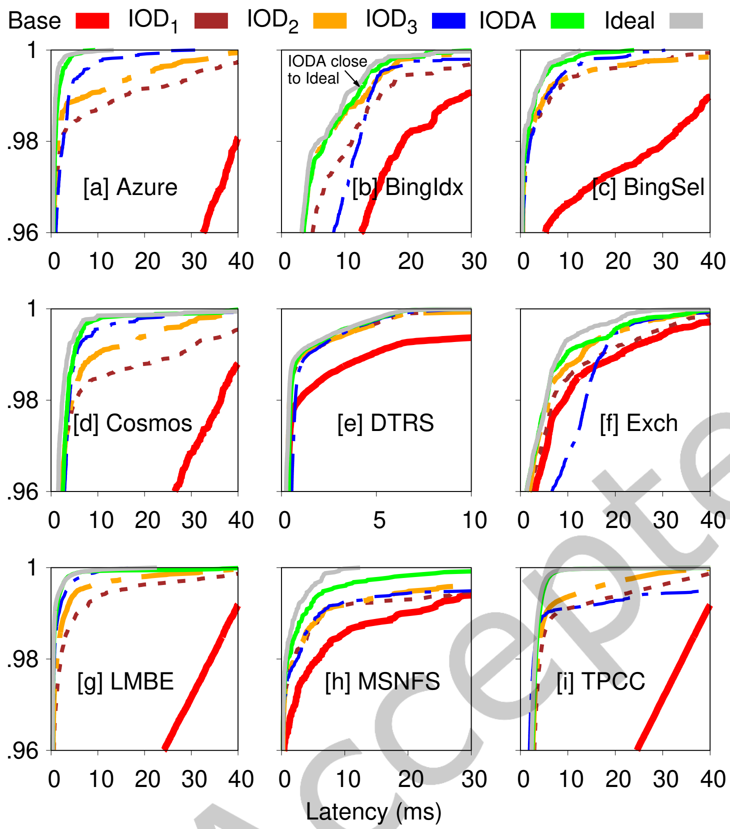


Fig. 6. **Read latency CDFs for all 9 block I/O traces (§6.1.2).** IODA *is the closest to the ideal case across all 9 block trace workloads.* IOD$_2$ *improves over* IOD$_1$*, but could not eliminate concurrent GC blockings.* IOD$_3$ *is worse than* IODA *due to whole-device level busy state.*

6.1.2 Many Workloads (Block I/O Traces). Figure 7 shows the p99 and p99.9 latencies with all the block traces. Figure 6 displays the complete read latency CDF graphs. [Key result #3] *Overall, with all these experiments with different workload characteristics and base latency distributions, the* IODA *bars in Figure 7 and CDF lines in Figure 6 summarize that* IODA *delivers faster latencies, 1.7× on average up to 16.3× between p95–p99.9 compared to the base approach, and only 1.0× to 3.3× slower than the* Ideal *case.*

Figure 8 shows the percentage of stripe-level reads that observe busy sub-IOs (from 1busy to 4busy), the top and bottom figures represent the percentage for the baseline and IODA, respectively. Similar to Figure 5b, it shows that IODA successfully shifts the concurrent GCs across time (higher 1busy green bars with almost no 2-4busy bars).

One small note is that in Figure 8b, we can see small IODA's 2-4busy bars for COSMOS and LMBE, but this is only 0.0005 of the time, due to a small implementation bug—upon further investigation, there are a very small number of left-over GCs that started just before and finished slightly after the time-window expires, which can be easily fixed in the future with a more precise time accounting.

6.1.3 File System, Key-Value, and Other Applications. We also ran various applications on ext4 on IODA, including 6 Filebench workloads, 3 YCSB/RocksDB workloads, and a dozen of data-intensive and stand-alone

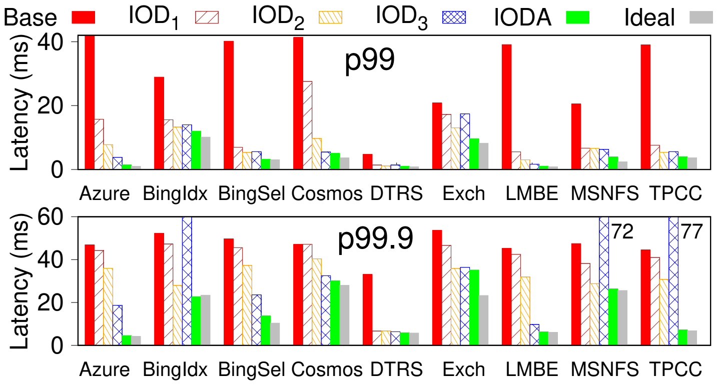


Fig. 7. **p99 and p99.9 latencies (§6.1.2).** *This figure details the p99 and p99.9 latencies from the I/O traces under all* IODA *strategies.* IODA *is the most deterministic and almost reaches the* Ideal *values.*

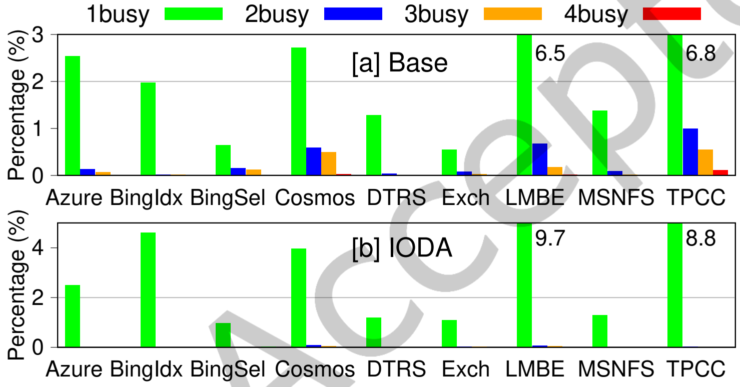


Fig. 8. **#Busy sub-IOs, many I/O traces (§6.1.2).** *The figure is the same type as Figure 5b, but now with many I/O traces.* IODA *shifts multiple concurrent* 2-4busy *sub-IOs (in the top* Base *figure) to more* 1busy *sub-IOs (in the bottom* IODA *figure).*

applications. The results are summarized in Figure 9, *all pointing to the same key conclusion that* IODA *is near to the ideal scenario.*

6.2 Versus State-of-the-Art Approaches

We now compare IODA with state-of-the-art approaches. For readability, this section mainly compares IODA with state-of-the-art approaches using one benchmark TPCC; other workloads show the same conclusion. All the results are aggregated in Figure 10.

6.2.1 vs. Proactive/Cloning (Always Full Stripe I/Os). A simple black-box way to cut 1-busy sub-IOs is to always proactively send a full-stripe read including the parity read (akin to cloning [3, 47, 93]), hence the I/Os can return to the user when the first $(N_{ssd}-k)$ sub-IOs finish. Figure 10a shows that Proactive is effective *but* still loses to IODA at high percentiles due to its inability to evade concurrent busy sub-IOs. Proactive also *negatively adds more load*. Figure 10b shows Proactive sends down 2.4× more I/Os than the base case, while IODA only issues 6% more reads.

Trace Workload	#I/Os (K)	Read/Write (%)	Read/Write (KB)	Max I/O (KB)	Interval (μs)	Size (GB)
`Azure`	320	18/82	24/20	64	142	5
`BingIdx`	169	36/64	60/104	288	697	11
`BingSel`	322	4/96	260/78	11264	2195	24
`Cosmos`	792	8/92	214/91	16384	894	63
`DTRS`	147	72/28	42/53	64	203	2
`Exch`	269	24/76	15/43	1024	845	9
`LMBE`	3585	89/11	12/191	192	539	74
`MSNFS`	487	74/26	8/128	128	370	16
`TPCC`	513	64/36	8/137	4096	72	25

Table 3. **Block I/O trace characteristics (§6.1).** *This table shows the detailed characteristics of the block traces we use. "#I/Os" denotes the total number of I/Os in the trace, "Read/Write (%)" means the percentage of read/write I/Os, and "Read/Write (KB)" shows the average read/write I/O size, "Max I/O (KB)" represents the maximum I/O size, and finally, "Interval (μs)" means the average inter-arrival time between two consecutive I/Os, "Size (GB)" refers to the total amount of data.*

6.2.2 vs. Synchronized GCs (e.g., Harmonia [34]). Synchronized GCs attempt to schedule the SSDs in an array to reduce GC impacts [34, 35, 51]. For example, Harmonia [34] manages the SSDs to perform GCs at the *same* time (*i.e.*, a localized slowdown is better than scattered ones). Figure 10c shows that Harmonia [34] improves the overall average latency by 27% compared to the baseline, *but* is far from achieving latency determinism due to the localized slowdown. IODA's alternating window strategy is more superior.

6.2.3 vs. Partitioning (e.g., Flash on Rails [36]). Flash on Rails (Rails) [36] partitions the SSDs such that user-vs-GC or user-vs-user contention is reduced. It divides an array into read-only and write-only SSDs, and performs read-write role swapping periodically. A similar strategy can also be found in Gecko [50] and SWAN [35]. Figure 10d shows that Rails is indeed able to deliver a pure read-only latency (the left-most orange line). The "raw" IODA (the right-most line) loses because *Rails relies on much NVRAM* to stage all inflight writes. In "raw" IODA, however, user reads are queued together with user writes. For a fair comparison, after we add a similar host-side write buffering, the $IODA_{NVM}$ line in Figure 10d shows roughly the same performance as Rails.

However, Rails has two *fundamental downsides: reduced throughput and requiring large NVRAM*. As SSDs are broken into read/write roles separately, there are *fewer* number of devices to serve reads (and writes). Figure 10e shows that Rails' throughput is significantly lower compared to IODA, underutilizing the array's bandwidth. Further, Rails requires much NVRAM to stage all incoming writes. The needed NVRAM is proportional to the write-mode duration and N_{ssd}, making it prohibitively too large for real systems.

6.2.4 vs. Preemptive GC. Preemptive GC (`PGC`) [26] is an approach that allows user reads to be interleaved in between GC individual read/write/erase operations, hence user reads are not queued far behind. Compared to the `Base` latency, `PGC` has already successfully reduced a huge area of the latency tail. But, the `IODA` line in Figure 10f shows that, vs. `PGC`, IODA is still more effective. This is because IODA users do *not* need to wait for *any* individual GC operation, but `PGC` users *sometimes* must wait for *at least one* individual GC operation.

6.2.5 vs. P/E Suspension. To further improve preemptive GC, more recent works suggest program/erase (P/E) suspension, even in the middle of a GC write/erase operation [29, 30]. It will deliver more stable latencies by allowing reads to "interrupt" write/erase and resume it later (see `Suspend` vs. `PGC` in Figure 10f). IODA still outperforms the suspension method.

A fundamental weakness of GC preemption and suspension is that these features *must be disabled* when the over-provisioning space is *full* (*e.g.*, under continuous write bursts). IODA's busy/predictable windows on the

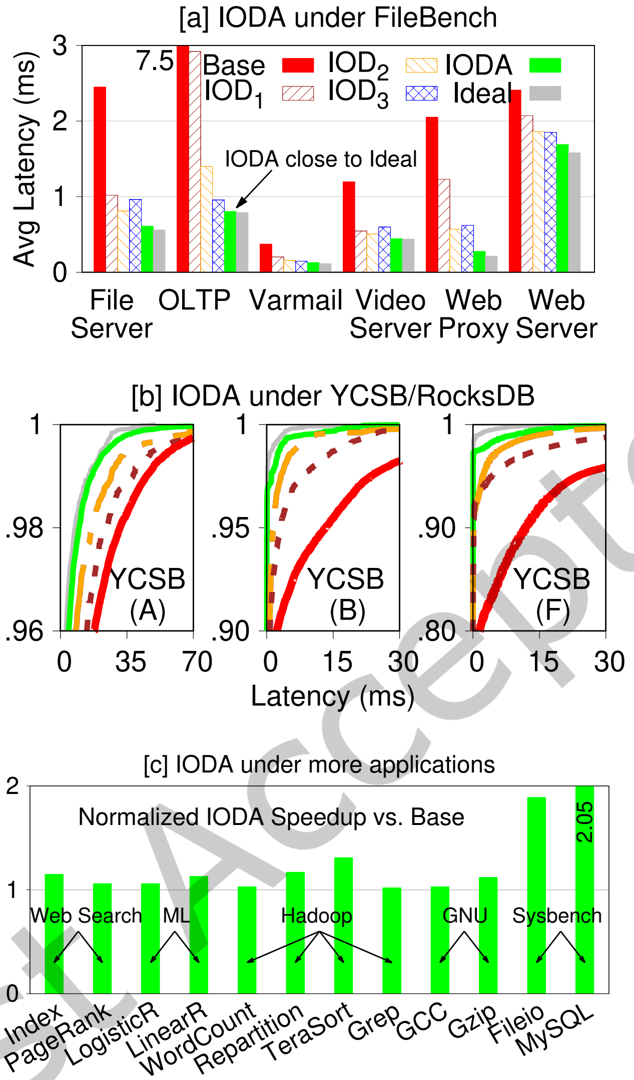


Fig. 9. **Filebench, YCSB, and other standalone/misc data-intensive application results (§6.1.3).** *Figure* **(a)** *shows the average latencies of six Filebench workloads as Filebench doesn't support per-IO latency logging.* `IODA` *is the most optimum and nearest to* `Ideal`*; Figure* **(b)** *presents latency CDFs for three YCSB workloads (A, B, and F), and again,* `IODA` *almost reaches the* `Ideal` *performance at high percentiles; Figure* **(c)** *shows the end-to-end normalized performance improvement (*`IODA` *vs.* `Base`*) based on workload-specific performance metrics (e.g., runtime, latency/ throughput, etc.)*

other hand alternate all the time. Figure 10g compares the performance of IODA and P/E suspension under a continuous maximum write burst. Here, we can clearly see that IODA's benefit is more apparent compared to the benefit of P/E suspension (the gap between the IODA and `Suspend` lines is larger in Figure 10g than in Figure 10f). **[Key result #4]** *Overall,* IODA *outperforms state-of-the-art methods in delivering deterministic latency, even under maximum write bursts.*

6.2.6 vs. TTFlash*.* TTFlash [10] is a "tiny-tail" flash controller design by pushing GCs to a finer-granularity (*i.e.*, chip level) and perform them rotationally. We followed TTFlash firmware organizations and implemented TTFlash logics in FEMU. Figure 10h shows IODA can achieve similar predictable latencies as a RAID-5 array of four TTFlash drives. However, TTFlash's internal RAIN [69] layout shrinks per-drive capacity and throughput as one channel is dedicated for in-device parity maintenance (25% degradation, not shown). We would also like to further stress that IODA design achieves predictable I/Os without heavily re-architecting the flash firmware/controller as TTFlash does (§3.3), thus distinguishing itself in its unique design context (host/device co-design), principles (simplicity for deployment), and technical challenges (PLM refinement and management, as well as host OS predictable I/O stack design).

6.2.7 vs. MittOS. MittOS [39] advocates a SLO-aware interface to allow quick I/O fail-over to replicas for fast response. It relies on "open/white-box" device knowledge to make OS-level predictions, thus not applicable for commercial devices. As shown in Figure 10i, MittOS loses to IODA as I/O fail-over might also be slow if the target node/device is busy. IODA's PL_{Win} approach eliminates the gap here. A side note, MittOS's I/O fast-rejecting interface is based on OS-level prediction to the underlying "profiled" devices, while IODA per-IO predictability flag (PL_{IO}) is lightweight and accurate with host/device collaboration.

6.3 Extended Evaluations

While previous sections focus on performance, this section covers other various aspects of IODA.

6.3.1 IODA *on OpenChannel-SSD (OCSSD).* IODA approach also runs well on real SSD hardware. We re-implement IODA's firmware changes in the Linux LightNVM driver ("host-side firmware") and run it on OCSSD [25]. Figure 10j shows a similar improvement as on FEMU, as shown earlier in Figure 5a.

6.3.2 IODA *on LightNVM on "$FEMU_{OC}$".* Unfortunately, our 5-year-old OCSSD became erratic and the vendor no longer supports/sells it; we could not complete more experiments on our OCSSD. This reality of real SSD hardware platforms is likely a reason why software-based flash emulators appear more recently in major venues [44, 45, 80, 81, 94]. Luckily, FEMU can also act as a drop-in replacement of OCSSDs for LightNVM [24] (a host-managed "$FEMU_{OC}$" with the device firmware stripped). Table 4 shows the normalized latency improvement of IODA vs. `Base` at major percentiles across various workloads.

6.3.3 IODA *on Commodity SSDs?* One might wonder whether IODA can be achieved on commodity SSDs without device-level modifications. We ran our *TW* algorithm, $\texttt{IOD}_3$ (PL_{Win}*-only* on the host side), on an array of real consumer SSDs. We set *TW* to 100ms, 1 and 10 seconds. Figure 10k shows that they are not effective (the three $\texttt{IOD}_3$ lines in red, brown and dashed blue) and far from the `Ideal` line as commercial SSDs do not have the PL_{IO} and PL_{Win} mechanism in place to kindly signal the host for proactive reconstructions. **[Key result #5]** *This experiment strongly shows the necessity to add small firmware modifications to honor the predictable latency mode window.*

	95^{th}	99^{th}	99.9^{th}	99.99^{th}
`Azure`	11.9	8.4	6.2	5.1
`BingIdx`	1.6	1.4	1.6	1.6
`BingSel`	3.7	3.1	2.3	1.9
`Cosmos`	9.2	5.6	1.8	1.4
`DTRS`	2.8	3.0	11.9	13.7
`Exch`	7.1	3.5	5.6	2.1
`LMBE`	16.0	8.0	1.9	1.3
`MSNFS`	1.4	2.8	12.1	6.3
`TPCC`	5.4	3.8	1.7	2.1
`YCSB-A`	7.3	3.1	3.5	4.7
`YCSB-B`	19.0	3.8	5.3	1.2
`YCSB-F`	6.8	4.4	7.2	5.4

Table 4. **IODA Speedup vs. Base** on top of $FEMU_{OC}$

6.3.4 IODA *Write Latency.* Back to the FEMU-based IODA, Figure 10l shows IODA benefits to write latencies. Each non-full-stripe write in RAID-5 triggers a read-modify-write process to update the parity page, hence user write latency is affected by the internal read performance. By virtue of predictable read latencies in IODA, write latencies are also significantly improved (up to p96 across all workloads, not shown). When user writes (or the associated parity updates) contend with device-level GCs, they might still get queued behind. That's the reason IODA write latency loses to "Ideal" for the last few percentiles.

6.3.5 IODA Throughput. Figure 11a shows the IODA and Base read/write IOPS under a 256-thread FIO benchmark with various read/write ratios (100/0, 80/20 and 0/100). Note that the IOPS is capped by FEMU's throughput (§5). One interesting phenomenon here is that IODA improves the write throughput by 9% in the 80/20 and 0/100 R/W configurations similarly because IODA improves the read latency in read-modify-write parity operations. For the same reason, read throughput increases by 10% in the 80/20 configuration. IODA does not degrade read throughput for the 100/0 case, showing minimal runtime system overhead.**[Key result #6]** IODA *does not sacrifice the raw RAID read/write throughput.*

6.3.6 Performance Sensitivity to TW Values. The current IODA setup uses TW=100ms based on the calculated value in Table 2 ("TW_{Burst}" row and "FEMU" column). Figure 11b shows the performance sensitivity under a smaller/larger TW value. The average load of the workload is ~13 DWPD (monitored at the device-level). If we calculate the TW value based on the TW formula in Figure 3, we get TW_{norm}=~5s, which is the upper-bound TW to guarantee the predictability contract. Under TW={500ms,2s} (*i.e.*, <5s), we can see that the green lines sticking together and all showing predictable latencies. However, if we further increase TW=10s, the SSDs *fail* to guarantee the absence of GC within the predictable windows, hence worse performance (*i.e.*, the SSDs couldn't reclaim enough space during the busy windows, and forceful/non-delayable GCs have to spawn into the predictable windows). This performance gap is more apparent in Figure 11c where we send a continuous *maximum* write burst that fills up the over-provisioned space faster. Under TW=20ms, we also see slightly worse performance (see "lower bound" in §4.3.2). As a result, some leftover disturbance is still felt after the device alternates to the predictable window.

6.3.7 Write Amplification Sensitivity to TW Values. To show the implication of various TW values to write amplification (WA), we ran a longitudinal analysis using an event-driven SSD simulator, SSDSim [95]. Figure 12 shows the result across different workloads and TW values. As expected, short windows (*e.g.*, 10ms) will cause high WA (*e.g.*, 1.2× or more) but long windows reduce the WA. Our 100ms busy window value for our emulated device delivers a reasonable WA (1.0 to 1.1× in most of the workloads). In Figure 12, compared to the Base case where no TW policy is applied, IODA's TW mechanism is equivalent to (safely) postponing GCs until one drive's turn to be unpredictable. This gives the firmware opportunities to absorb more user writes before starting GCs on victim blocks with fewer valid pages to move. Thus, GC efficiency is higher and that's why we see WA decreases for almost all the workloads as TW increases (X-axis). As discussed in §4.3.4, operators can use a practical DWPD value to increase window durations and reduce WA further.

6.3.8 Re-configuring TW for better WA. As discussed in §4.3.7, flash array operators could dynamically adjust the TW for their target workloads to balance WA and predictability (*i.e.*, use TW=TW_{norm} instead of TW_{burst}). In Figure 13, we ran 3 synthetic FIO workloads with different write intensities (40, 80, and 20 DWPD) each for one hour. For each workload, we configure IODA to use TW=TW_{burst} for the first 30 minutes and TW=TW_{norm} for the second half. We report the p99.9 latencies (every 10 minutes) and write amplification factor (WA). From the top and middle figures in Figure 10, we can see that IODA can sustain predictable latencies while improving WA by switching to a larger TW.

6.3.9 Other evaluations. IODA also works for RAID-4/6. When running TPCC, IODA is only 1-1.16× slower than the Ideal case between p95–p99.99 while the Base is 3-4.9× slower. We also benchmarked IODA under a wider array (N_{ssd}=8/16), and achieved similar results. We omit the detailed results for space.

7 DISCUSSION

7.1 PL_{Win} for Coordinated SSD Buffer Flush

SSD internal buffers need to be frequently flushed (*i.e.*, writing cached items to the backend NAND flash), and this will potentially cause heavy contention between user reads and flush-caused writes when the buffer space runs critical. As a result, user reads will suffer from long latencies when scheduled behind flush operations. One way to solve this problem is adapting our PL_{Win} approach to the buffer eviction scenario. The idea is to force SSDs in the same array to do extensive buffer flushing in a coordinated manner, as demonstrated in Figure 2 but for flush operations, not GCs.

7.2 Host-Managed Dynamic TW

With the previous window value calculations, the users (*e.g.*, RAID controller) still need to rely on SSD vendors to calculate and advertise the window length. Ideally, the host should be able to manage the windows without breaking the I/O determinism contract. In this approach, we do not add more work to the SSD, but rather have the host dynamically set the window value. Upon reboot, the host sets a base value B (*e.g.*, $0.5sec$) and during runtime dynamically adjusts the value using a simple algorithm below, which is only *possible* given the busy signals supported in IODA, hence showing the power of all of the approaches combined.

Every period of P (*e.g.*, 50ms), the host increases the value by I ms (*e.g.*, 10ms) as long as it does not see more than k busy sub-IOs within a stripe (k=1 in RAID-5). In other words, as long as the host can always reconstruct up to k busy sub-IO(s) within a stripe, then the window value is deemed "safe," as it allows all the SSDs to have enough time to perform background operations without overlapping each other in time. As mentioned earlier (§4), ideally, PL_{Win} is set as high as possible to reduce write amplification, hence the reason we increase the value gradually.

If more than k busy sub-IOs are observed, it implies that PL_{Win} is too large for the current user load, hence forcing the SSDs to execute some background operations within the supposedly deterministic period and breaking the IOD expectation. For example, the internal RAM buffer or the over-provisioned NAND space is almost full, forcing a flush or GC to happen, respectively. When the host sees more than k busy sub-IOs, the host decreases the PL_{Win} by half[3] and informs the SSDs of the new window value.

We acknowledge that there are potentially many other possibilities to set the window value. For example, device performance likely deteriorates over time, thus even the static method requires window time recalibration. Above are our early attempts to figure out ways to program the window value and we find them simple and effective enough.

7.3 Vendor Willingness for IODA Interface Changes

The current IOD-PLM specification already suggests that devices expose some lower and upper bound of the predictable time window (but again no literature discusses how to program them).

IODA extensions do *not* reveal much information about the device's proprietary internals. First, the PL_{BRT} technique can be made optional. Our earlier approach (§4) is powerful enough and can be combined with the next method where PL_{BRT} only helps in very corner cases, as explained later. Second, we argue that returning PL_{BRT} does not reveal more information beyond what users already see. Prior works already show that users can deconstruct many internal SSD layouts by simply deconstructing the user-observed latencies [97–99]. Third, if slight "obfuscation" is needed, PL_{BRT} can be designed to be a normalized number to alleviate potential timing channel attacks, similar to the chunk wearing information in the OpenChannel 2.0 specification [100]. PLM also suggest upper bound. And this kind of "gray-box information" [101, 102] is valuable because it does not reveal the

[3] Mimicking the TCP AIMD algorithm (additive increase multiplicative decrease) [96].

	SSD#0	SSD#1	SSD#2	SSD#3	SSD#4	SSD#5	SSD#6	SSD#7
0N–1N	×	.	.	.	.	.	.	.
1N–2N	.	×	.	.	.	.	.	.
2N–3N	.	.	×	.	.	.	.	.
3N–4N	.	.	.	×	.	.	.	.
4N–5N	.	.	.	.	×	.	.	.
5N–6N	.	.	.	.	.	×	.	.
6N–7N	.	.	.	.	.	.	×	.
7N–8N	.	.	.	.	.	.	.	×

Table 5. **More fine-grained time window (§7.4).** *This table shows an example of a more fine-grained time window mechanism based on SSD LBA ranges. Here, the top row represents the 8 SSDs (#0–#7) in the same flash array, and each* $[iN..(i+1)N]$ *represents the LBA range from* iN *to* $(i+1)N$. *"×" represents background operations are allowed to happen in the SSD while "." means background operations are disallowed.*

internal details but yet is helpful for the host. Guessing the remaining time in a black-box way will be challenging due to the many vendor-specific implementations (different FTLs, GC algorithms, etc.).

Overall, IODA only changes the interface minimally wihtout exposing SSD proprietary details.

7.4 Fine-Grained Time Window (TW)

As a future direction, one can make IODA TW implementation *even* more fine grained. In existing IODA design, an SSD is not allowed to perform GCs during the busy TW. Even with predicability-flagged I/Os (PL_{IO}), it only helps the host to query device side busyness, but not scheduling time windows (TW) at a more fine-grained level (*e.g.*, channel level). The question is whether we could allow GCs to happen during busy TW but breaking IODA's strong predictability guarantees.

It is important to note is that concurrent GCs that delay pages in different stripes are tolerable. For example, consider two full-stripe I/Os A and B that each will create seven parallel pages to seven SSDs ($A_1..A_7$ and $B_1..B_7$). It is possible that a GC in SSD#0 blocks A_1 and another concurrent GC in SSD#1 blocks B_2. Let us assume one parity per stripe (r=1). As long as parities A_8 and B_8 are not blocked, IODA can tolerate the two GCs as they delay pages in different stripes. This is the reason why IODA can tolerate r delayed pages per I/O stripe. So what we can do is to provide a **2 dimensional** time window. For example, SSD#0 is allowed to do GC from LBA [0 to N), but other SSDs are not allowed to GC on LBA [0 to N). At the same time window, SSD#1 is allowed to do GC from LBA [N to $2N$), but other SSDs are not allowed to do it. In other words in every time window we have a 2-dimensional configuration where the x-axis is the SSD numbers and the y-axis is the logical partitioning of the LBAs (which we can configure). For example, if we break the per-SSD LBA into 4 logical partition, each with N bytes: $0 - N, N - 2N, 2N - 3N, 3N - 4N$, the configuration will be like Table 5: From Table 5, we can see that SSD#0 is only allowed to do GC within 0N-1N LBAs, and so on. In the next few time windows, we slide the configuration accordingly. The advantage of this approach is that the SSDs are all still doing GCs at the same time.

However, this more fine-grained approach is more challenging to implemenet. Let's say to utilize full parallelism, L1 are mapped to row#1, and L2 is mapped to row#2. It works well when we GC data in L2, we can use a copyback mechanism where data in L1 and L2 does not leave the chip, hence we won't have any contention. But these days GC copyback is not enabled, because during GC the SSD piggyback ECC checking to check that data is valid. In this case, the controller must read the data via the channel to the DRAM, hence the channel will be busy and content with the user I/Os. We leave further exploration as future work.

8 CONCLUSION

To the best of our knowledge, IODA is a host/SSD co-designed flash array that provides a strong latency predictability contract without sacrificing the aggregate bandwidth. IODA only involves minimal changes to the NVMe interface and flash firmware to simplify deployment. IODA delivers close-to-ideal latencies and outperforms many state-of-the-art approaches. We hope IODA will spur more work around the new and exciting IOD-PLM interface.

REFERENCES

[1] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP)*.

[2] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the Killer Microseconds. *Commun. ACM* 60, 4 (2017).

[3] Jeffrey Dean and Luiz Andre Barroso. 2013. The Tail at Scale. *Communications of the ACM (CACM)* 56, 2 (2013).

[4] 2018. Why Deterministic Storage Performance is Important. https://www.architecting.it/blog/deterministic-storage-performance/. (2018).

[5] 2020. All-Flash NVMe Reference Architecture. https://www.samsung.com/semiconductor/global.semi/file/resource/2020/05/redhat-ceph-whitepaper-0521.pdf. (2020).

[6] 2020. Micron 9100 U.2 and HHHL NVMe PCIe SSDs. https://www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/9100_hhhl_u_2_pcie_ssd.pdf. (2020).

[7] 2021. Achieve Consistent Low Latency for Your Storage-Intensive Workloads. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/low-latency-for-storage-intensive-workloads-article-brief.html. (2021).

[8] Ross Stenfort, Ta-Yu Wu, and Lee Prewitt. 2020. NVMe Cloud SSD Specification. https://www.opencompute.org/documents/nvme-cloud-ssd-specification-v1-0-3-pdf. (2020).

[9] 2020. Storage Latency in Flash Arrays. https://www.violinsystems.com/wp-content/uploads/Storage-Mojo-WP-storage-latency.pdf. (2020).

[10] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs, See [108].

[11] Nima Elyasi, Changho Choi, Anand Sivasubramaniam, Jingpei Yang, and Vijay Balakrishnan. 2019. Trimming the Tail for Deterministic Read Performance in SSDs. In *IEEE International Symposium on Workload Characterization (IISWC)*.

[12] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. 2014. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[13] 2016. GreyBeards on Storage. https://silvertonconsulting.com/gbos2/tag/tail-latency/. (2016).

[14] Chris Petersen, Wei Zhang, and Alexei Naberezhnov. 2018. Enabling NVMe I/O Determinism @Scale. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180807_INVT-102A-1_Petersen.pdf. (2018).

[15] Kapil Karkra. 2018. Using Software to Reduce High Tail Latencies on SSDs. https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2018/20180808_SOFT-201-1_Karkar.pdf. (2018).

[16] 2020. Data Set Management Commands Proposal for ATA8-ACS2. http://www.t13.org/Documents/UploadedDocuments/docs2008/e07154r6-Data_Set_Management_Proposal_for_ATA-ACS2.pdf. (2020).

[17] 2020. NVM Express Base Specification 1.0. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_0e.pdf. (2020).

[18] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully Automatic Stream Management for Multi-Streamed SSDs Using Program Contexts. In *Proceedings of the 17th USENIX Symposium on File and Storage Technologies (FAST)*.

[19] 2020. NVM Express Base Specification 1.4. https://nvmexpress.org/wp-content/uploads/NVM-Express-1_4-2019.06.10-Ratified.pdf. (2020).

[20] Jon C. R. Bennett. 2012. Memory Management System and Method. https://www.google.com/patents/US8200887. (2012).

[21] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[22] Yaochen Hu, Yushi Wang, Bang Liu, Di Niu, and Cheng Huang. 2017. Latency Reduction and Load Balancing in Coded Storage Systems. In *Proceedings of the 8th ACM Symposium on Cloud Computing (SoCC)*.

[23] Heiner Litz, Javier Gonzalez, Ana Klimovic, and Christos Kozyrakis. 2021. RAIL: Predictable, Low Tail Latency for NVMe Flash. *ACM Transactions on Storage (TOS)* 1, 1 (2021).

[24] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator, See [105].

[25] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. LightNVM: The Linux Open-Channel SSD Subsystem, See [108].

[26] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, Feiyi Wang, and Jongman Kim. 2011. A Semi-Preemptive Garbage Collector for Solid State Drives. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[27] 2014. Pre-emptive Garbage Collection of Memory Blocks. https://www.google.com/patents/US8626986. (2014).

[28] Junghee Lee, Youngjae Kim, Galen M. Shipman, Sarp Oral, and Jongman Kim. 2013. Preemptible I/O Scheduling of Garbage Collection for Solid State Drives. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*.

[29] Guanying Wu and Xubin He. 2012. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension, See [104].

[30] Shine Kim, Jonghyun Bae, Hakbeom Jang, Wenjing Jin, Jeonghun Gong, Seungyeon Lee, Tae Jun Ham, and Jae W. Lee. 2019. Practical Erase Suspension for Modern Low-latency SSDs, See [106].

[31] 2015. Erase Suspend/Resume for Memory. https://patents.google.com/patent/US9223514B2/en. (2015).

[32] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[33] Suzhen Wu, Weidong Zhu, Guixin Liu, Hong Jiang, and Bo Mao. 2018. GC-aware Request Steering with Improved Performance and Reliability for SSD-based RAIDs. In *Proceedings of the 32th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

[34] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. 2011. Harmonia: A Globally Coordinated Garbage Collector for Arrays of Solid-state Drives. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*.

[35] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays, See [106].

[36] Dimitris Skourtis, Dimitris Achlioptas, Noah Watkins, Carlos Maltzahn, and Scott Brandt. 2014. Flash on Rails: Consistent Flash Performance through Redundancy. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*.

[37] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs, See [108].

[38] Jaeho Kim, Donghee Lee, and Sam H. Noh. 2015. Towards SLO Complying SSDs Through OPS Isolation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*.

[39] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. 2017. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*.

[40] 2014. MZHPV128HDGM (SM951) 128 GB PCIe Gen3 8Gb/s x4 M.2. http://www.samsung.com/semiconductor/products/flash-storage/client-ssd/MZHPV128HDGM. (2014).

[41] Chun-Yi Liu, Jagadish Kotra, Myoungsoo Jung, and Mahmut T. Kandemir. 2018. PEN: Design and Evaluation of Partial-Erase for 3D NAND-Based High Density SSDs, See [105].

[42] Michael Mesnier, Jason B. Akers, Feng Chen, and Tian Luo. 2011. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*.

[43] George Amvrosiadis, Angela Demke Brown, and Ashvin Goel. 2015. Opportunistic storage maintenance. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*.

[44] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, and Myoungsoo Jung. 2018. FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[45] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. 2021. Prolonging 3D NAND SSD Lifetime via Read Latency Relaxation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[46] Katherine Missimer and Richard West. 2018. Partitioned Real-Time NAND Flash Storage. In *Proceedings of the 39th IEEE Real-Time Systems Symposium (RTSS)*.

[47] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: Cutting Tail Latency in Cloud Data Stores via Adaptive Replica Selection, See [103].

[48] Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-Effective Redundancy for Lower Latency Variance on Cloud Storage Services, See [103].

[49] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S. Gunawi. 2020. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network, See [111].

[50] Ji-Yong Shin, Mahesh Balakrishnan, Tudor Marian, and Hakim Weatherspoon. 2013. Gecko: Contention-Oblivious Disk Arrays for Cloud Storage. In *Proceedings of the 11th USENIX Symposium on File and Storage Technologies (FAST)*.

[51] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. 2014. Coordinating Garbage Collection for Arrays of Solid-State Drives. *IEEE Transactions on Computers (TC)* 63, 4 (April 2014).

[52] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. 2009. Gordon: using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[53] Feng Chen, Rubao Lee, and Xiaodong Zhang. 2011. Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing. In *Proceedings of the 17th International Symposium on High Performance Computer Architecture (HPCA-17)*.

[54] Myoungsoo Jung, Wonil Choi, Miryeong Kwon, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut Kandemir. 2019. Design of a Host Interface Logic for GC-Free SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 8, 1 (May 2019).

[55] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[56] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *Proceedings of the 19th USENIX Symposium on File and Storage Technologies (FAST)*.

[57] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. 2014. Willow: A User-Programmable SSD, See [107].

[58] Sungjin Lee, Ming Liu, SangWoo Jun, Shuotao Xu, Jihong Kim, and Arvind. 2016. Application-Managed Flash. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*.

[59] Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. De-indirection for Flash-based SSDs with Nameless Writes, See [104].

[60] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Greg R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*.

[61] Amy Tai, Igor Smolyar, Michael Wei, and Dan Tsafrir. 2021. Optimizing Storage Performance with Calibrated Interrupts. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[62] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. 2020. DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation. In *Proceedings of the 18th USENIX Symposium on File and Storage Technologies (FAST)*.

[63] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters, See [109].

[64] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[65] Fay W. Chang and Garth A. Gibson. 1999. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[66] Christian Navasca, Cheng Cai, Khanh Nguyen, Brian Demsky, Shan Lu, Miryung Kim, and Guoqing Harry Xu. 2019. Gerenuk: thin computation over big native data using speculative program transformation. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*.

[67] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. 2014. The Power of Choice in Data-Aware Cluster Scheduling, See [107].

[68] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the Outliers in Map-Reduce Clusters using Mantri. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[69] 2019. Redundant Array of Independent NAND for a Three-dimensional Memory Array. https://patents.google.com/patent/US20170249211A1/en. (2019).

[70] Martin Maas, Krste Asanovic, Tim Harris, and John Kubiatowicz. 2016. Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[71] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. 2015. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[72] Nanqinqin Li, Mingzhe Hao, Huaicheng Li, Tim Emami, and Haryadi S. Gunawi. 2022. Fantastic SSD Internals and How to Learn and Use Them. In *Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR)*.

[73] Joonsung Kim, Pyeongsu Park, Jaehyung Ahn, Jihun Kim, Jong Kim, and Jangwoo Kim. 2018. SSDcheck: Timely and Accurate Prediction of Irregular Behaviors in Black-Box SSDs. In *51st Annual IEEE/ACM International Symposium on Microarchitecture*

(MICRO-51).
[74] 2020. What's the State of DWPD? Endurance in Industry Leading Enterprise SSDs. http://www.storagesearch.com/dwpd.html. (2020).
[75] 2015. Speeds, Feeds and Needs – Understanding SSD Endurance. https://blog.westerndigital.com/ssd-endurance-speeds-feeds-needs/. (2015).
[76] 2021. Non-Volatile Random-Access Memory. https://en.wikipedia.org/wiki/Non-volatile_random-access_memory. (2021).
[77] 2021. Intel Optane Persistent Memory (PMem). https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. (2021).
[78] 2021. IODA Github Homepage. https://github.com/huaicheng/IODA. (2021).
[79] 2018. FEMU Github Homepage. https://github.com/ucare-uchicago/femu. (2018).
[80] Yun-Sheng Chang, Yao Hsiao, Tzu-Chi Lin, Che-Wei Tsao, Chun-Feng Wu, Yuan-Hao Chang, Hsiang-Shang Ko, and Yu-Fang Chen. 2020. Determinizing Crash Behavior with a Verified Snapshot-Consistent Flash Translation Layer, See [111].
[81] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs, See [110].
[82] Roman Pletka, Ioannis Koltsidas, Nikolas Ioannou, Sasa Tomic, Nikolaos Papandreou, Thomas Parnell, Haralampos Pozidis, Aaron Fry, and Tim Fisher. 2018. Management of Next-Generation NAND Flash to Achieve Enterprise-Level Endurance and Latency Targets. In *ACM Transactions on Storage (TOS)*.
[83] Open-Channel Solid State Drives. http://lightnvm.io/. ([n. d.]).
[84] Cosmos+ OpenSSD Platform. http://openssd.io/. ([n. d.]).
[85] DFC Open Source Community. https://github.com/DFC-OpenSource. ([n. d.]).
[86] 2017. Emulab D430s. https://gitlab.flux.utah.edu/emulab/emulab-devel/wikis/Utah-Cluster/d430s. (2017).
[87] 2020. Ultra-Low Latency with Samsung Z-NAND SSD. https://www.samsung.com/us/labs/pdfs/collateral/Samsung_Z-NAND_Technology_Brief_v5.pdf. (2020).
[88] 2016. SNIA I/O Trace Data Files. http://iotta.snia.org/traces. (2016).
[89] Filebench. https://github.com/filebench/filebench/wiki. ([n. d.]).
[90] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*.
[91] 2020. Sysbench. https://github.com/akopytov/sysbench. (2020).
[92] 2020. HiBench: The Bigdata Micro Benchmark Suite. https://github.com/Intel-bigdata/HiBench. (2020).
[93] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective Straggler Mitigation: Attack of the Clones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.
[94] Myungsuk Kim, Jisung Park, Geonhee Cho, and Yoona Kim. 2020. Evanesco: Architectural Support for Efficient Data Sanitization in Modern Flash-Based Storage Systems, See [110].
[95] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance Impact and Interplay of SSD Parallelism through Advanced Commands, Allocation Strategy and Data Granularity. In *Proceedings of the 25th International Conference on Supercomputing (ICS)*.
[96] Additive increase/multiplicative decrease. https://en.wikipedia.org/wiki/Additive_increase/multiplicative_decrease. ([n. d.]).
[97] Laura M. Grupp, John D. Davis, and Steven Swanson. 2013. The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*.
[98] Aviad Zuck, Philipp Guhring, Tao Zhang, Donald E. Porter, and Dan Tsafrir. 2019. Why and How to Increase SSD Performance Transparency. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*.
[99] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The Bleak Future of NAND Flash Memory, See [104].
[100] Open-Channel Solid State Drives Specification (Revision 2.0). http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf. ([n. d.]).
[101] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. 2001. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*.
[102] Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2004. Deploying Safe User-Level Network Services with icTCP, See [109].

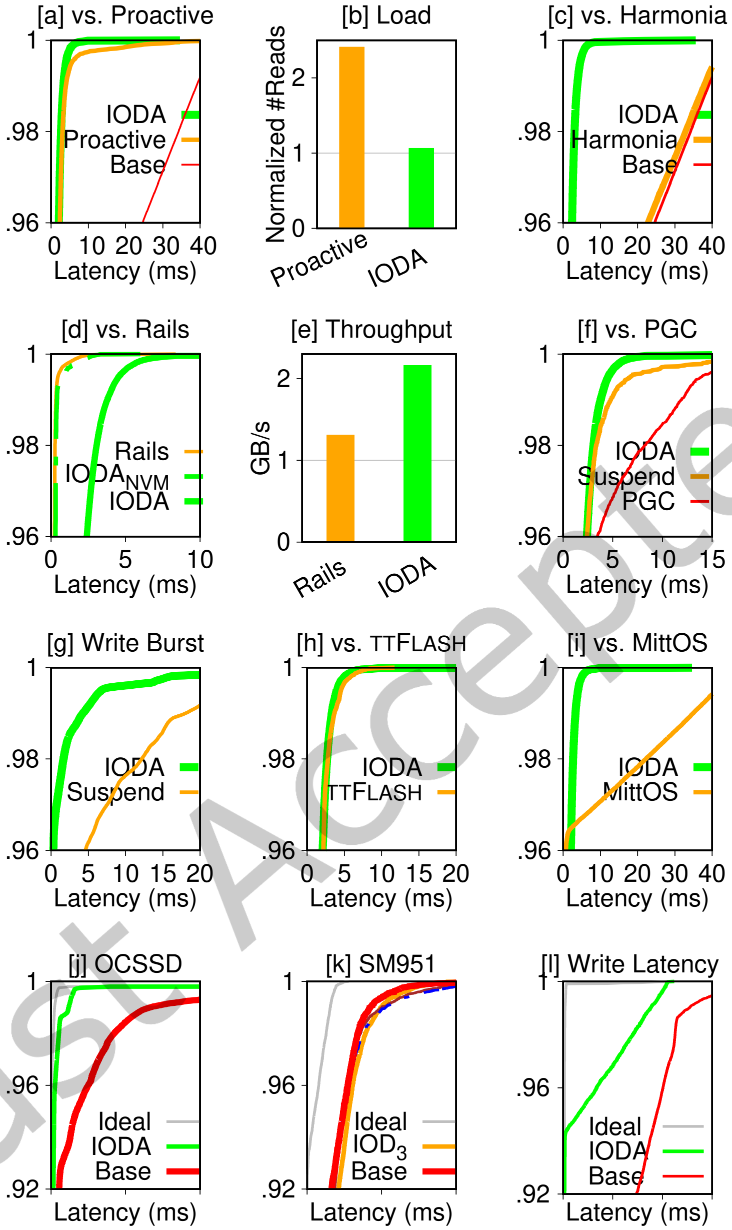


Fig. 10. **IODA vs. 7 state-of-the-art approaches (§6.2) and extended evaluations (§6.3).** *In Figure* **(a)–(i)**, IODA *outperforms almost all the 7 competing approaches in delivering predictable I/Os without sacrificing array bandwidth, burdening the system with excessively extra load, or requiring excessive host-side buffering or device-side changes. Figure* **(j)–(l)** *show* IODA *extended evaluations on OpenChannel-SSD (OCSSD) and commercial (SM951) SSDs, and write latency:* ***(j)*** IODA *achieves predictable latencies on a real OCSSD (§6.3.1),* ***(k)*** *how unmodified commodity SSDs requires our proposed device-level modifications (§6.3.3).* ***(l)*** IODA *improves write latencies by virtue of improved read latencies for the read-modify-write parity update process (§6.3.5).*

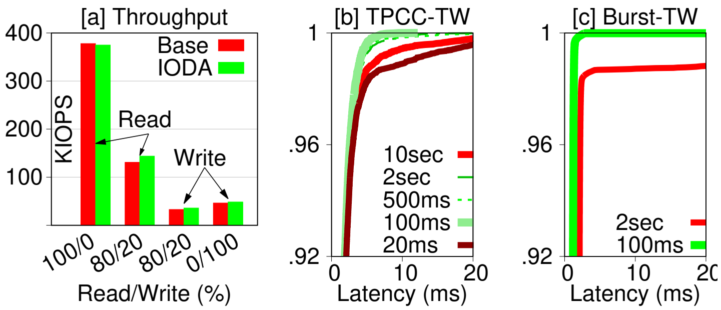


Fig. 11. **IODA throughput and performance sensitivities to TW.** ***(a)*** *IODA-vs-Base read/write throughput under various read/write ratios (§6.3.5),* ***(b)*** *TW sensitivity on TPCC performance (§6.3.6),* ***(c)*** *same as (b) but under maximum write burst (§6.3.6).*

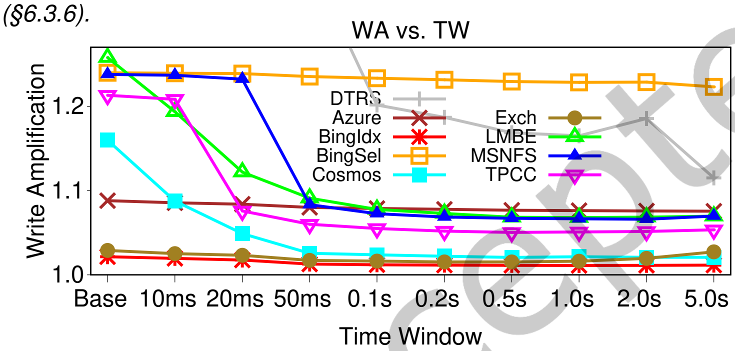


Fig. 12. **Write amplification sensitivity (§6.3.7).** *The y-axis shows the write amplification factor (WAF) and the x-axis varies the TW value.*

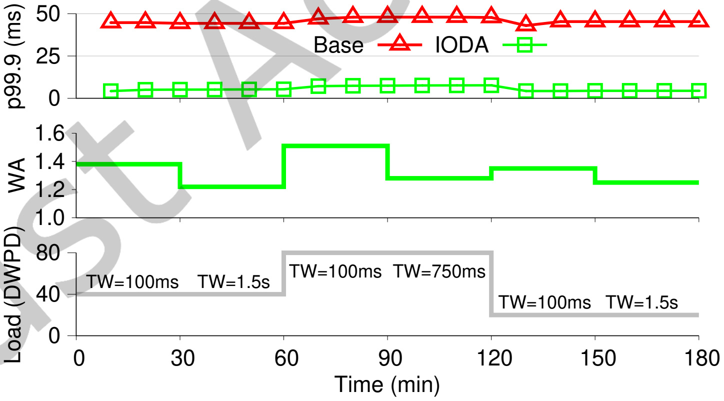


Fig. 13. **Adjusting TW for predictability and low WA (§6.3.8).** *This figure shows how the host can reconfigure TW to achieve low WA without sacrificing latency predictability.*