IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage

Huaicheng Li
University of Chicago and Carnegie Mellon University

Martin L. Putra
University of Chicago

Ronald Shi
University of Chicago

Xing Lin
NetApp

Gregory R. Ganger
Carnegie Mellon University

Haryadi S. Gunawi
University of Chicago

Abstract

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, 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 95–99.9th percentiles by up to 75×. IODA is also the nearest to the ideal, no disturbance case compared to 7 state-of-the-art preemptive, suspension, GC coordination, partitioning, tiny-tail flash controller, prediction, and proactive approaches.

CCS Concepts

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

1 Introduction

Flash arrays are popular storage choices in data centers and they must address users’ craving for low and predictable latencies [1–3]. Thus, many recent SSD products are released and evaluated not just on the average speed but the percentile latencies as well [4–7]. 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 [8–10]. 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) [11] to provide legroom for more efficient 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 [12–14].

To tame the SSD performance challenges, there have been many efforts to evolve the device interfaces [15–17]. The Stor-
age Interface Technical Committee has standardized many extensions over the last decade: from UNMAP/TRIM (2011) [15], ATOMIC_WRITE (2013) [16], STREAM (2017) [17], to a recent one, the NVMe I/O Determinism (IOD) interface (2019) [18]. 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. 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. 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 ($B_0 = B_2 \oplus B_1 \oplus P$), 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” [9, 19–22], 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. To this end, we introduce IODA\(^1\), 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 [23, 24], 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 (§5) 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 [25–27], program/erase suspension [28–30], speculation [31, 32], GC coordination [33, 34], partitioning [35–37], “tiny-tail” controller design [9], and SLO-aware prediction [38].

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 throughputs; 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).

\(^1\)IODA is pronounced “Yoda,” a wise and determined Jedi Master.

2 IOD-PLM: The Good and The Better

2.1 How IOD-PLM Works

The NVMe I/O Determinism (IOD) concept [18] 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.

### 2.2 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 [18, §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.

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.

Third, the PLM busy window duration is vital for a strong predictability guarantee (more in §3.3), however, we are not aware of any work that attempts to analyze and formalize 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.

### 2.3 Related Work and Our Contributions

Table 1 summarizes existing approaches that attack the flash performance challenges. The popular methods include preemptions [25, 28, 39], hints [38, 40–43], partitioning [10, 36, 37, 44], speculation [21, 45, 46], latency prediction [38, 47], and coordinated GCs [9, 33–35, 48, 49]. Traditional preemptions cannot indefinitely avoid/postpone GCs, as they will revert to normal blocking behavior under insufficient over-provisioning space. Hint-schemes such as [43] require code changes, breaking application transparency. Partitioning methods like FlashBlox [36] 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 [2] 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 [38] or LinnOS [47] answer the when-to-reconstruct question but suffer from inaccuracies without collaboration with the device. Coordinated GCs, as in TTFLASH [9], 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 [9, 25, 42, 50–52], host-level changes [31, 34, 35, 53, 54], transparent approaches on programmable devices [11, 24, 38, 55, 56], to interface solutions [17, 49, 57–59]. 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) [58] 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 [9] 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) 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 [13, 60], 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 §5.2, qualitatively and quantitatively.

<table>
<thead>
<tr>
<th>Determinism</th>
<th>Throughput</th>
<th>Transparency</th>
<th>Deployment</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 1:** Comparison of IODA to state-of-the-art approaches. IODA achieves performance determinism without sacrificing throughputs, and is transparent to applications with minimal device-side changes for easy deployment.
While these existing works without a doubt guide us to our ultimate solution, 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? Our unique contributions lie in answering the above questions.

3 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 3.4 puts all the pieces together.

3.1 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 rearchitected. 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.

3.2 PLIO: Predictable-Latency Flagged I/Os

Our first method is to introduce PLIO, 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. PLIO binary response serves as a timely and accurate signal for the host to initiate proactive reconstruction. PLIO 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=\text{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=\text{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 (§2). 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 on (PL=false (00)), which we call reconstruction I/Os to differentiate from the original user I/Os. After reconstruction, the host can return the user I/Os to the upper layers (e.g., file systems) and deem it completed.

3.2.1 Benefits and Limitations. This simple extension delivers a large benefit for two reasons. First, failing an I/O only takes 1\text{u}s through PCIe and the xor-based reconstruction takes less than 10\text{u}s 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-I/Os of the same stripe are delayed by simultaneous GCs on different devices is significantly lower than the probability of just one sub-I/O getting delayed. A sub-I/O is a page I/O within a full-stripe I/O. In a 4-drive array, a full-stripe I/O has 4 sub-I/Os, 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-I/Os 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-I/Os are not predictable (i.e., the reconstruction I/Os also cannot be served quickly). The subsequent sections will address this limitation and show how PLIO can be more powerful under further enhancement.
3.2.2 A Further Extension (PLBRT). To address the limitation of PLIOD, 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-I/Os 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 PLBRT improves upon PLIOD, but PLBRT fails to provide a strong predictability contract. The PLBRT 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-level GC operations kick in at relatively the same time. PLBRT becomes ineffective here because the host would see multiple unpredictable I/Os with similar busy remaining time values.

3.3 PLWin: Busy Latency Windows

3.3.1 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 a 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 1 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 \(t+(2 \times TW)\), 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 (i), 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.

While similar coordination ideas as in Figure 1 have appeared in scenarios ranging from in-device RAIN [9, 19, 61] to even distributed “Java GC” [62, 63], 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 PLWin, 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 [58]).

3.3.2 TW Upper-Bound Formulation. This section describes our TW formulation in a top-down fashion. To satisfy
\[ TW \leq \frac{R_p \times S_t}{(N_{ssd} \times \text{Min}(B_{pcie}, \text{Max}(\frac{N_{dwpd} \times (1 - R_p) \times S_t}{8\text{hours/day}}))) - (\frac{(1 - R_o) \times N_{ch} \times S_{pg} \times N_{pg}}{(t_r + t_w + 2 \times t_{cpt}) \times R_o \times N_{pg} + t_e})} \]

Figure 2: The TW formulation. The formula depends on 11 hardware-level parameters and 3 workload related parameters (the width of the flash array \(N_{ssd}\), \(R_o\), \(N_{dwpd}\)). The breakdown of the formula is in Table 2.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(t_{cpt})</td>
<td>TimeOfChannelPageTransfer</td>
<td>(\mu s)</td>
<td>40 60 40 60 60</td>
</tr>
<tr>
<td>(t_w)</td>
<td>TimeOfNandPageWrite</td>
<td>-</td>
<td>1440 140 960 2000 1940</td>
</tr>
<tr>
<td>(t_r)</td>
<td>TimeOfNandPageRead</td>
<td>(\mu s)</td>
<td>60 40 40 32 60 50</td>
</tr>
<tr>
<td>(t_e)</td>
<td>TimeOfNandBlockErase</td>
<td>ms</td>
<td>8 3 3 3 6 3</td>
</tr>
<tr>
<td>(B_{pcie})</td>
<td>BandwidthOfPCIE</td>
<td>GB/s</td>
<td>4 8 4 4 8 8</td>
</tr>
</tbody>
</table>

### Hardware Time Specification

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_{pg})</td>
<td>SizeOfNandPage</td>
<td>KB</td>
<td>16 16 4 16 16 16</td>
</tr>
<tr>
<td>(N_{pg})</td>
<td>NumberOfPagesPerBlock</td>
<td>-</td>
<td>512 512 256 384 256 256</td>
</tr>
<tr>
<td>(N_{blk})</td>
<td>NumberOfBlocksPerChip</td>
<td>-</td>
<td>2048 2048 256 2731 5461 4096</td>
</tr>
<tr>
<td>(N_{chip})</td>
<td>NumberOfChipsPerChannel</td>
<td>-</td>
<td>4 8 4 8 8 8</td>
</tr>
<tr>
<td>(N_{ch})</td>
<td>NumberOfChannels</td>
<td>-</td>
<td>8 16 8 8 12 16</td>
</tr>
<tr>
<td>(R_p)</td>
<td>RatioOfOverProvisioning</td>
<td>-</td>
<td>0.25 0.12 0.25 0.20 0.40 0.20</td>
</tr>
<tr>
<td>(R_o)</td>
<td>RatioOfGCValidPages</td>
<td>-</td>
<td>0.5 0.75 0.75 0.75 0.75 0.75</td>
</tr>
</tbody>
</table>

### Derived Values

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(S_{blk})</td>
<td>SizeOfNandBlock</td>
<td>MB</td>
<td>8 8 1 6 4 4</td>
</tr>
<tr>
<td>(S_t)</td>
<td>SizeOfTotalNandSpace</td>
<td>GB</td>
<td>512 2048 16 512 2048 2048</td>
</tr>
<tr>
<td>(S_p)</td>
<td>SizeOf ProvisionSpace</td>
<td>GB</td>
<td>128 246 4 102 819 410</td>
</tr>
</tbody>
</table>

### Garbage Collection

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{gc})</td>
<td>TimeToGCOneBlock</td>
<td>ms</td>
<td>658 617 57 312 425 408</td>
</tr>
<tr>
<td>(S_{gc})</td>
<td>SizeOfGCReclaimedSpace</td>
<td>MB</td>
<td>32 32 2 12 12 16</td>
</tr>
<tr>
<td>(B_{gc})</td>
<td>BandwidthOfGCCleaning</td>
<td>MB/s</td>
<td>49 52 35 38 28 39</td>
</tr>
</tbody>
</table>

### Storage Specification

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N_{dwpd})</td>
<td>NumberOfCommonDWP</td>
<td>-</td>
<td>10 10 40 10 10 10</td>
</tr>
<tr>
<td>(B_{norm})</td>
<td>BandwidthOfWorkloadWrite</td>
<td>MB/s</td>
<td>137 641 17 146 437 582</td>
</tr>
<tr>
<td>(B_{burst})</td>
<td>BandwidthOfFullWrite</td>
<td>MB/s</td>
<td>3200 4000 536 3200 3204 4000</td>
</tr>
</tbody>
</table>

### RAID

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N_{ssd})</td>
<td>NumberOfSSDsInTheArray</td>
<td>-</td>
<td>8 4 4 8 4 4</td>
</tr>
</tbody>
</table>

### Time Window

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Longest Symbol</th>
<th>Unit</th>
<th>Symbol Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T_{norm})</td>
<td>TimeWindowNormal</td>
<td>ms</td>
<td>6259 5014 6206 4622 24380 9171</td>
</tr>
<tr>
<td>(T_{burst})</td>
<td>TimeWindowBurst</td>
<td>ms</td>
<td>256 790 97 204 327 1315</td>
</tr>
</tbody>
</table>

Table 2: Time window (TW) breakdown and values (§3.3). 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).

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 1 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 ($Sp$) divided by the net write load, hence the constraint above. Given that $Sp$ 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 $TW$.

### 3.3.3 $TW$ Parameters

We now break down our $TW$ formulation in a bottom-up fashion. The parameters we introduced ($Sp$, $B_{burst}$ and $B_{gc}$) are high-level parameters that must be derived from hardware specifications. Figure 2 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 ($t_{chn}$), 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 ($Sp$), pages per block ($N_{pg}$), blocks per chip ($N_{hbk}$), 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 ($Sp_{hbk}$), total NAND size ($S_{t}$), and over-provisioning space ($Sp$). 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 ($N_{ssd}$).

### 3.3.4 $TW$ 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”) [23], an OpenChannel-SSD (“OCSSD”) [24] whose parameters are publicly known, and 3 commercial SSDs from different vendors. We used an SSD prober 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 [64]. The average number of valid pages in a victim block ($R_{v}$) is estimated from running our workloads ($\S$5 in FEMU [23] and OpenChannel-SSD [24]. We emphasize that the use of these numbers are only for analyzing possible $TW$ values on real devices. Optionally, a more detailed version of Table 2 can be found in the Appendix (Table 2), covering all the driven values for the $TW$ parameters.

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 predictable window length, 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.

### 3.3.5 $TW$ Scalability & Write Amplification (WA)

We now analyze the trade-offs of $TW$ values. Figure 3a 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 (1x$TW$). This means the over-provisioning space will be full relatively faster.

Unfortunately, as shown in Figure 3b, 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 $\S$3.3.7). 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.

### 3.3.6 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) [65]. For calculating $B_{norm}$ in Table 2, we use DWPD values of 10 to 40 ($N_{dwpd}$), often suggested to prolong

---

**Figure 3: 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 $TW$.}
the device lifetime to 3-5 years [66].

Plugging in this value to the same formula will give us \( TW_{\text{norm}} \). As shown in the two last rows of Table 2, \( TW_{\text{norm}} \) increases the busy window length by 6-64x (higher compared to \( TW_{\text{burst}} \)), 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 \( B_{\text{norm}} \) 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.

3.3.7 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 3c 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 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 \( T_{\text{igc}} \), and peaks around \( TW=TW_{\text{burst}} \), 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 \( TW \in [TW_{\text{burst}}, TW_{\text{40DWP}}] \), or \( [TW_{\text{burst}}, TW_{\text{20DWP}}] \), respectively. Here, \( TW_{\text{40DWP}} \) represents the calculated \( TW \) value based on Figure 2 with \( N_{\text{4DWP}}=40 \). Combining the red line WA trend, the flash array operators better switch the \( TW \) from \( TW_{\text{burst}} \) to \( TW_{\text{40DWP}} \) for better WA if the workload intensity decreases from “Burst” to “40DWP”. 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.

3.4 Putting It All Together

In summary, we show that the two combinations of \( \text{PL}_{\text{IO}} + \text{PL}_{\text{Win}} \) creates a very efficient flash array that fulfills the two rules of the strong contract we mentioned in Section 3.3. When not combined, each of these two techniques has limitations (which we will evaluate later).

**PL\text{IO} only:** As discussed before, this method advocates a “fail-if-slow” hardware design to enable host-level timely re-construction for better latencies. However, it does not prevent multiple sub-I/Os within a stripe from concurrent GC delays in different SSDs, thus the inability to achieve predictable latency when multiple SSDs are busy.

**PL\text{Win} only:** Although \( \text{PL}_{\text{Win}} \) 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 \( \text{PL}_{\text{Win}} \) 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\text{IO} + PL\text{Win}):** When \( \text{PL}_{\text{IO}} \) and \( \text{PL}_{\text{Win}} \) are combined, the host will always send I/Os with \( \text{PL}=\text{true} \) (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 \( \text{PL}_{\text{Win}} \) 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 9b, and with <10% fast-rejected reads in Figure 7 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 \( \text{PL}_{\text{BRT}} \) (the shortest-background-remaining-time strategy), as stated in Section 3.2, we no longer need it but will still evaluate it for SSD vendors who do not prefer SSDs to be array-aware. (In \( \text{PL}_{\text{Win}} \), the \( TW \) calculation requires the SSDs knowing \( N_{\text{sd}} \), the number of devices).

**Interface and control flow:** To achieve \( \text{PL}_{\text{IO}}+\text{PL}_{\text{Win}} \), 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) \( \text{arrayType} \) and (2) \( \text{arrayWidth} \). Next, the device plugs in these values to program \( TW \) internally and returns the value via (3) \( \text{busyTimeWindow} \) 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 1).

**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 §3.3 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 [67] or Optane Memory [68]) 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 “nd” 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. 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).

### 4 Implementation

We now describe IODA implementation [69].

**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 [23, 70] used by some recent works appearing in top venues [42, 43, 71, 72]. To make FEMU resemble modern SSDs, we had to make several optimizations in 1200 LOC. 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. (b) **Light-NVM+OCSSD.** We also prototype IODA with LightNVM on a real OCSSD [24, 73] 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.

**IODA’s host side:** The host-side logic is written in 1814 LOC in Linux 4.15 Software RAID (i.e., the Linux nd sub-system) 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/Raid/Reather I/O PL-flag passing, and the complex Linux RAID 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 [25, 29, 31, 33, 35] in around 3400 LOC.

### 5 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 (§5.1). Then we present comparisons of IODA with the state of the art (§5.2) and show extended evaluations (§5.3).

**Platform setup:** Most experiments are done on FEMU (for reasons mentioned in §4) running on Emulab D430 machines [74]. 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. 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 [75]) 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 (BingSel1) and 5 standard SNIA block traces [76] that we have re-rated 8-32 times more intense to reflect mod-
ern 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 [77] and 3 YCSB/RocksDB workloads [78] on the ext4 file system. Additionally, we also run 12 other storage workloads ranging from GNU applications, Sysbench [79], to MapReduce (Hadoop/Spark) workloads [80]. 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 YYth percentile.

5.1 Main Results
This section shows the improvement made by the combination of IODA strategies, one at a time: “IOD1” represents only predictable-latency flagged I/Os (PL=only in §3.2), “IOD2” the shortest-busy-remaining-time strategy (PL=only in §3.3), “IOD3” the alternating busy windows only (PL=WIN in §3.3, without PL=only/PL=BRT), and “IODA” the final approach (PL=WIN+PL=WIN as described in §3.4). For IOD3 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.

5.1.1 IODA Techniques, 1 Workload First. For figure simplicity, we first show only the results of using one workload, TPCC (Table 3). Figure 4a 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 IOD1 line shows that by just circumventing the busiest (slowest) read, via proactive data reconstruction as signaled by the PL=only method, the latency is more predictable up to p99. (2) The orange IOD2 line shows that the PL=BRT busy-remaining-time approach further helps but cannot completely evade concurrent busyness. (3) The blue IOD3 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.

Figure 4: IODA percentile latencies and #busy sub-I/Os with TPCC (§5.1.1). 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-I/Os (x-axis).

Figure 5: Read latency CDFs for all 9 block I/O traces (§5.1.2). IODA is the closest to the ideal case across all 9 block trace workloads. IODA improves over IOD1, but could not eliminate concurrent GC blockings. IOD3 is worse than IODA due to whole-device level busy state.

Figure 6: p99 and p99.9 latencies (§5.1.2). This figure details the p99 and p99.9 latencies from different I/O traces under all IODA strategies. IODA is the most deterministic and almost reaches the Ideal values.
(§3.4). (4) [Key result #1] Finally, the bold green IODA line in Figure 4a shows that PL_{IO+PW} 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 4b reveals the reason behind IODA’s success. The x-axis shows how many “sub-IOs” of a stripe are returned busy (PL=11, §3.2). 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 IODA_{1} and IODA_{2} can reconstruct 1 busy sub-IO, it cannot evade this concurrent busyness. That is why the IODA_{1} and IODA_{2} lines in Figure 4a start increasing between the p99 and p99.9 values. [Key result #2] With our final approach, the green IODA bar in Figure 4b shows that our time-window approach successfully shifts concurrent GCs across time such that at any time there is at most only 1 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).

5.1.2 Many Workloads (Block I/O Traces). Figure 6 shows the p99 and p99.9 latencies with all the block traces. Figure 5 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 6 and CDF lines in Figure 5 summarize that IODA delivers faster latencies, 1.7x on average up to 16.3x between p95–p99.9 compared to the base approach, and only 1.0x to 3.3x slower than the Ideal case.

Figure 7 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 4b, it shows that IODA successfully shifts the concurrent GCs across time (higher 1busy green bars with almost no 2-4busy bars).

5.1.3 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 applications. The results are summarized in Figure 8, all pointing to the same key conclusion that IODA is near to the ideal scenario.

5.2 Versus 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 9.

5.2.1 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 [2, 45, 81]), hence the I/Os can return to the user when the first (N_{ext}−k) sub-IOs finish. Figure 9a 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 9b shows Proactive inefficiently sends down 2.4x more I/Os to the base case, while IODA only issues 6% more reads.

5.2.2 vs. Synchronized GCs (e.g., Harmonia [33]). Synchronized GCs attempt to schedule the SSDs in an array to reduce GC impacts [33, 34, 49]. For example, Harmonia [33] manages the SSDs to perform GCs at the same time (i.e., a localized slowdown is better than scattered ones). Figure 9c shows that Harmonia [33] 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.

5.2.3 vs. Partitioning (e.g., Flash on Rails [35]). Flash on Rails (Rails) [35] 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 [48] and SWAN [34]. Figure 9d 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 in-flight writes.
Figure 8: Filebench, YCSB, and other standalone/misc data-intensive application results (§5.1.3). 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.)

Figure 9: IODA vs. 7 state-of-the-art approaches (§5.2) and extended evaluations (§5.3). 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 (§5.3.1), (k) how unmodified commodity SSDs requires our proposed device-level modifications (§5.3.3). (l) IODA improves write latencies by virtue of improved read latencies for the read-modify-write parity update process (§5.3.5).

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 \( IODA_{NVMe} \) line in Figure 9d 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 9e 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.

5.2.4 vs. Preemptive GC. Preemptive GC (PGC) [25] 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 9f 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.

5.2.5 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 [28, 29]. It will deliver more stable latencies by allowing reads to “interrupt” write/erase and resume it later (see Suspend vs. PGC in Figure 9f). IODA outperforms the suspension method.

A fundamental weakness of GC preemption and suspension is that these features must be disabled when the over-
This experiment strongly shows provisioning space is full (e.g., under continuous write bursts), IODA's busy/predictable windows on the other hand alternate all the time. Figure 9g 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 9g than in Figure 9f). **[Key result #4]** Overall, IODA outperforms state-of-the-art methods in delivering deterministic latency, even under maximum write bursts.

**5.3.6 vs. TTFLASH.** TTFLASH [9] 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 9h shows IODA can achieve similar predictable latencies as a RAID-5 array of four TTFLASH drives. However, TTFLASH's internal RAIN [61] 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 (§2.3), 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).

**5.2.7 vs. MittOS.** MittOS [38] 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 9i, 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.

---

**5.3 Extended Evaluations**

**5.3.1 IODA on OpenChannel-SSD (OCSSD).** IODA approach also runs well on real SSD hardware. We re-implement IODA’s firmware changes in the Linux Light-NVM driver (“host-side firmware”) and run it on OCSSD [24]. Figure 9j shows a similar improvement as on FEMU, as shown earlier in Figure 4a.

**5.3.2 IODA on Light-NVM on “FEMUOC”.** 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 [42, 43, 71, 72, 82]. Luckily, FEMU can also act as a drop-in replacement of OCSSDs for Light-NVM [23] (a host-managed “FEMUOC” with the device firmware stripped). Table 4 shows the normalized latency improvement of IODA vs. Base at major percentiles across various workloads.

**5.3.3 IODA on Commodity SSDs?** One might wonder whether IODA can be achieved on commodity SSDs without device-level modifications. We ran our \( TW \) algorithm, \( IOD_{TT} \) (\( PL_{Win-only} \) on the host side), on an array of real consumer SSDs. We set \( TW \) to 100ms, 1 and 10 seconds. Figure 9k shows that they are not effective 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.

**5.3.4 IODA Write Latency.** Back to the FEMU-based IODA, Figure 9l 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.

**5.3.5 IODA Throughput.** Figure 10a 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 (§4). 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. [Key result #6] IODA does not sacrifice the raw RAID read/write throughput.

### 5.3.6 Performance Sensitivity to TW Values

The current IODA setup uses TW=100ms based on the calculated value in Table 2 (“TWburst” row and “FEMU” column). Figure 10b 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 2, we get TWnorm=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 10c 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 §3.3.2). As a result, some leftover disturbance is still felt after the device alternates to the predictable window.

### 5.3.7 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 [83]. Figure 11 shows the result across different workloads and TW values. As expected, short windows (e.g., 10ms) will cause high WA (e.g., 1.2x or more) but long windows reduce the WA. Our 100ms busy window value for our emulated device delivers a reasonable WA (1.0 to 1.1x in most of the workloads). As discussed in §3.3.4, operators can use a practical DWPD value to increase window durations and reduce WA further.

### 5.3.8 Re-configuring TW for better WA

As discussed in §3.3.7, flash array operators could dynamically adjust the TW for their target workloads to balance WA and predictability (i.e., use TW=TWnorm instead of TWburst). In Figure 12, 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=TWburst for the first 30 minutes and TW=TWnorm 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 Conclusion

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.

### 7 Acknowledgments

We thank Mark Silberstein (our shepherd), and the anonymous reviewers for their tremendous feedback and comments. For University of Chicago authors, this material was supported by NSF grants CCF-1336580, CNS-1526304, CNS-1405959, CCF-2028427) as well as generous donations from Dell EMC and NetApp Faculty Fellowship. We also thank the member companies of the PDL Consortium (Amazon, Facebook, Google, HPE, Hitachi, IBM, Intel, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Seagate, Two Sigma, and Western Digital) and VMware for their interest, insights, feedback, and support. We extend our special thanks to Michael Hao Tong, Jaeyoung Do, Achmad Imam Kistijantoro, Fadhil I. Kurnia, Sujin Park, and Mingzhe Hao for their initial involvement and contributions to this project.
References


