Fantastic SSD Internals and How to See/Use Them

Submission No. 216

Abstract

This work presents Queenie, an application-level tool that can extract 10 internal properties from any block-level SSDs, together with the analysis result of running Queenie on 21 different SSD models from 7 major SSD vendors — Kelpie, which, to the best of our knowledge, is the the most comprehensive SSD probing study. By bringing numerous observations and unique findings, Kelpie exposes substantial improvement space for both SSD users and manufacturers, enlightening possibilities of unleashing more performance potential in real world scenarios and highlighting the necessity of further exploring our SSD internals.

1 Introduction

Solid-State Drives (SSDs) are becoming the cornerstone of modern storage systems because of their competitive performance, reliability, capacity and cost [4, 12, 17, 28, 33, 37]. However, while fulfilling user's increasing demands on storage, modern SSDs also bring their own challenges: they are difficult to be optimally-utilized as most of them show up as blackbox devices, with internal complexities such as FTL mapping, write buffer management and garbage collection mechanisms, hidden and intangile from their users [19, 21, 24, 39, 40].

These complexities, unfortunately, can bring non-negligible side-effects, with performance inconsistency as one of the notorious ramifications. For example, write buffer flush can contend with reads on NAND resources and bring long latency tails [3, 10, 17]; Reads with inappropriate alignment can take extra overhead to process as SSDs apply minimal unit of access [23, 25]; Some SSDs are designed for certain purposes, and when used inappropritately, can dramatically downgrade the overall system performance [6, 27].

Motivated to resolve these negative impacts, multiple works [11, 23, 25, 26, 27] try to extract crucial SSD properties and propose coherent designs based on the observations. These works have reasonably argued that probing SSDs can help build more effective solutions and bring significant performance improvement.

Based on these gains, we further ask: is there more knowledge hidden in modern SSDs, waiting to be mined and utilized, especially as modern SSDs have evolved rapidly. We found that there are many questions not answered in prior works. Do modern SSDs have favorable sizes on reads and punish those that do not comply (§4.1,§5.1)? Do components that were prevalent in SSDs previously, such as read buffer, still exist in current SSDs (§4.5)? Do large capacity SSDs, which are very common nowadays, have write buffer of appropriate sizes (§4.2) and capability to handle high-parallelism writes (§4.4)? Do SSDs apply hybrid (externally and internally triggered) buffer flush policies (§4.3) that can be exploited for less contention and better performance (§5.2)? Do SSDs really perform better when they face less "stress" (§4.6)?

To answer those questions and to provide a holistic tool that can extract many internal SSD parameters, in this paper, we introduce Queenie and Kelpie[1].

Queenie is a application-level tool that uses 18 core utility functions to probe the SSDs with carefully tuned read/write mixed workloads. By just measuring latencies observable from applications, Queenie attempts to extract 10 SSD internal properties and can run on any block-device SSDs.

Kelpie represents our analysis result of running Queenie on 21 different SSD models from 7 major SSD vendors, from regular consumer-level ones to latest enterprise-level SSDs. Kelpie bring numerous observations and 6 unique findings, exposing substantial improvement space for both SSD users and manufacturers. To show how Kelpie is important, we perform case studies on how and to what extent would these discoveries affect SSD performance and give suggestions regarding the possibilities of unleashing more performance potential in real world scenarios.

The paper's contribution/organization is as follows:

1. A long exposition of SSD internal properties that can be extracted and why such knowledge is important to applications (§2).
2. Queenie, an application-level SSD-probing tool that can extract many internal SSD properties. Queenie will be open-sourced as we are not aware of a similar tool publicly available (§3).
3. Kelpie, a thorough study covering 10 SSD properties on 21 SSD models (§4, 6), to the best of our knowledge, the most comprehensive SSD probing study. We will release Kelpie's 10 GB raw data.
4. Case studies on how applications can leverage this knowledge in in real-world scenarios (§5).
5. A discussion and conclusion that more "Newt"s[1] are needed to solve the complexities, intricacies, and nuances of modern state-of-the-art SSDs (§7).

[1] In the "Fantastic Beasts and Where to Find Them" movie, **Queenie** is a character who can read other people's minds, **Kelpie** is a shape-shifting creature that can take any form, representing the dynamic nature of our findings which depend on the probed SSD models, and **Newt** is a wizard who help solve crimes such as the crimes of Grindelwald in the second sequel.

2 Goals

In this section, we expand the goals of our work by describing the important properties to probe (§2.1), the target SSD models (§2.2), and the uniqueness of our work compared to prior works (§2.3).

2.1 Properties and Advantages

To design Queenie, we must first decide the important SSD internal properties that we try to extract. Below, we provide the list, the definition of each of the properties and the advantages of knowing them, as summarized in Table 1.

P_1. **Page size** is the simplest property, the minimal unit of read and write. Knowing the page size is the foundation of further probing techniques. *Advantage*: Knowing the internal page size will help applications align IOs properly to avoid unnecessary alignment overhead such as read-modify-write [23, 25]. Unfortunately, later, we show that this simple practice is not well enforced nowadays (§4.1).

P_2. **Page type** represents the NAND cell type (SLC, MLC, TLC, etc.) and how the page offsets are mapped to low/medium/high bits of the MLC/TLC cells. A page offset that is mapped to higher bits tend to have a higher latency. *Advantage*: Mapping hot data to low pages (lower latency) can bring performance improvement [15] under a typical static logical-to-physical page offset mapping at the NAND block level [40].

P_3. **Chunk size** reveals the striping unit inside the device (similar to RAID chunk size). For example, if an SSD has 16 chips, it might spread incoming 16 pages evenly among the 16 chips (1-page chunk), or it could also split them into 2 groups to 2 chips with 8 pages each (8-page chunk). *Advantage*: This information can help applications understand the throughput of sequential reads/writes.

P_4. **Stripe width** is the number of chunks that can be parallelized internally without contention at the chip level (akin to RAID stripe width). *Advantage*: Knowing this parallelism level allows applications to exploit the internal bandwidth better, *e.g.*, how databases are redesigned to map better to the SSD internal paralellism [11].

P_5. **Channel/chip layout** represents the number of channels and chips per channel. *Advantage*: Though sometimes treated as "boring" fact, this property can unearth unusual findings. We found that channel/chip layout is not always symmetrical. For example, a channel can exhibit more contention (higher latency) with some channels than others (§6).

P_6. **Read performance consistency** is about whether the SSDs have favorable read sizes and "punish" those who do not follow, *e.g.*, whether SSDs can process non-paged, sectored reads with minimal alignment overhead and whether large reads are broken into smaller ones and served simultaneously. *Advantage*: It is worthy to double check these standard expectations, as sometimes we find exceptional cases (§4.1) that might require special handling (§5.1).

ID	Name	Output Format
P_1	Page size	Size (*e.g.*, 4 KB)
P_2	Page type	S/M/TLC + low/high
P_3	Chunk size	Size (*e.g.*, 64 KB)
P_4	Stripe width	A number (*e.g.*, 64)
P_5	Channel/chip layout	#Channels * #Chips
P_6	Read perf consistency	'Good'(✓) or 'Bad'(✗)
P_7	Read buffer cap	A size or none
P_8	Write buffer cap	A size or none
P_9	Write parallelism	A number (*e.g.*, 4)
P_A	Internal flush window	A duration (*e.g.*, 5ms)
P_B	Garbage collection (GC) interval	A number (*e.g.*500 writes)

Table 1: **Properties covered by existing works + ours (§2.1).** *This table shows the SSD properties (later would be referred to by ID, i.e. P1-PA) covered by existing works plus ours.*

P_7. **Read buffer capacity** is about guessing how much of the internal RAM is occupied for caching reads. As NAND read speed gets faster and faster, we check whether vendors still employ read cache inside the device. *Advantage*: Caching is paramount to performance. Knowing this information can hint of a better cache design at a higher layer.

P_8. **Write buffer capacity** similarly is about speculating how much the internal RAM is used for buffering user writes before flushing them to the NAND cells. *Advantage*: Write buffer flush is a dominant factor of when garbage collection (background noise) will kick in. Having the knowledge of the size, together with controlling/tracking user write operations, can help predict the timing of flushes and potential garbage collection activities [27].

P_9. **Write parallelism** is the number of parallel writes supported by the device. The issue here is that unlike read operations, writes tend to be absorbed to a centralized buffer first. This probing checks whether SSDs are able to maintain high write parallelism regarless of the centralized buffering. *Advantage*: The result here can bring insights on how to apply SSD-specific optimizations for write workloads, while revealing defects on write handling in some SSDs (§4.4).

P_A. **Internal flush window** is the amount of time the device needs to softly dump entire write buffer to the underlying NAND without causing severe contention. Besides externally being forced to trigger a flush (*e.g.*, an almost full buffer due to write bursts), SSDs also perform background flushes periodically during idle time or when the load is low. *Advantage*: As this internal flush usually incurs less contention and shorter blocking time, it can be further exploited to design a solution that mitigates the buffer flush interference.

P_B. **Garbage collection (GC) interval** is about predicting when GCs will happen, for example, in the form of a statistical distribution of #writes between adjacent GCs. *Advantage*: Arguably, this information is one of the most secretive but valuable ones. For example, if applications can

	#	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_A	P_B
[25]	4		✓					✓	✓			
[11]	2	✓		✓	✓							
[27]	7								✓			$\frac{1}{2}$
Q&K	21	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Table 2: **Queenie vs. existing works.** *This table compares Queenie and Kelpie (Q&K) with other SSD probing works. '#' stands for number of SSD models probed by each work. As we can see, Queenie covers more SSDs and properties.*

know when GC is happening, they can avoid it by re-routing read IOs to other less-busy replicas [16]. We are not aware of prior works that successfully and affirmatively unveil this information. Some tried but was not conclusive enough [27]. Similary, we tried but more future work remains [2].

2.2 SSD Models

Our goal is to probe a wide variety of SSDs from different vendors such that we can constrast the probing results from different devices. From our labs and other industry partners, we have amassed 21 SSD models from 7 vendors, from consumer-level ones (8 pieces) to enterprise-level SSDs (13). Their interfaces vary: SATA (7), SAS (6) and NVMe (8). Their sizes range from 64 GB to 2 TB, covering different NAND technologies such as SLC (6), MLC (12), and TLC (3). The full list of our devices can be found later in Table 3 in Section 6. As mentioned, we are not aware of any other probing papers with this scale of hardware being probed.

2.3 Related Work

We briefly discuss related work on SSD and storage probing, SSD performance modeling and host-managed SSDs.

SSD probing: There are existing works that attempt to probe SSD internals [11, 25, 27], but as shown in Table 2, our work provides more contributions. First, Queenie and Kelpie cover more properties ($P_1 - P_A$), catching more "fantastic" facts of modern devices. Second, Queenie has been tested on 21 SSD models including enterprise ones, while other works only probe up to 7 consumer devices. Another note is we do not cite other earlier works [23, 26] as they are shorter articles of those listed in the table.

Storage probing. Pulling up one more level, storage probing in general is a common area for decades, for example, probing HDDs [35, 38], memory hierarchy [41], RAID [13], SMR [5], and USB drives [8]. These works introduce the positive outcomes of better understanding the hardware internals. However, their mechanisms cannot be easily ported to SSDs given the different physical nature. As we did not find many works in extensive SSD probing (Table 2), we decided to perform one.

SSD performance modeling: A different, complementary way of direct probing is black-box performance modeling [9, 14, 18, 20, 30, 31]. These works show that modeling SSDs is feasible by just collecting external performance metrics. Others do warn that the models could be error prone as they "may not necessarily apply to other SSD models" [43].

Host-managed SSDs: We acknowledge the rise of software-defined SSDs to increase controllability either via extended interfaces [22, 29] or full exposure of SSD internals [7, 32, 42]. Probing is only useful for black-box commodity SSDs, which is our focus here given the larger scale of deployment of such SSDs.

3 Queenie

We now describe Queenie probing strategies (§3.1) and then briefly showcase the core algorithms (§3.2).

3.1 Probing Methods

Queenie probes the 10 properties ($P_1 - P_A$) as follows. LPN and PPN represent logical and physical page numbers.

P_1. **Page size**: Queenie extracts the smallest unit of read by reading continuous 0.5KB sectors to detect interval between page borders. For example, assume an SSD has page size of 4KB and page borders at 0, 4 and 8KB, then a read of 2 sectors at offset 3KB would require SSD to read only one page, while the same read at 3.5KB will require SSD to read 2 pages with *higher* latency, indicating a page border at 4KB. By repeating similar reads at larger offsets, Queenie will see another adjacent border at 8KB, thus confirming the 4KB page size. The probing function here is named `F1_pushRead`. Below, all mentions of "4KB" implies the page size obtained by the `F1_pushRead` function.

P_2. **Page type**: Queenie first erases and sequentially writes the SSD. This will give us some kind of a static mapping as SSDs are unlikely randomizing the LPN-PPN mapping for every page. Queenie then sends large 4KB sequential reads *one page at a time* and compare the read latency of each page. To get away with the side effects of internal readahead (if any), we read from higher to lower page offsets (readahead usually begins caching when seeing monotonically increasing back-to-back offsets). For M/TLC drives, we will observe *different* latencies as the offsets *vary*, as pages are mapped to diffferent lower/medium/higher bits of the cell. With this, Queenie can get the LPN positions of low/medium/high pages. The function here is "`F2_rangeRead`." If the SSD performs a static mapping (common these days), these LPN-to-low/medium/high mapping will remain the same. One can re-run `F2_rangeRead` (without the erase) occasionally to confirm that.

P_3. **Chunk size**: Queenie reuses `F1_pushRead` to detect chunk borders by reading consecutive pages. With an SSD of 16-page chunk size and chunk borders at 0, 16, and 32 pages, then a 2-page read at page 14 will go to only one chip, while reading at page 15 would be served by two chips in parallel with *lower* latency, indicating a border at page 16.

Similarly, by increasing the offset, Queenie will see another border at page 32 and infer chunk size of 16 pages.

P_4. **Stripe width**: For this, Queenie issues multiple concurrent chunk reads with incremental offsets. For instance, for an SSD with stripe width of 16 chunks, issuing 4 reads with an offset increment of exactly the stripe width would cause these 4 reads sent to the same chip, resulting in heavier contention than those with smaller offset increments. The function to probe this is defined as `F3_strideRead`.

P_5. **Channel/chip layout**: `F3_strideRead` can also help detect the overall channel/chip layout by sending multiple concurrent page reads with both chunk-level and stripe-width offset increments, which will show us more contention to confirm how many channels and #chips/channel exist.

P_6. **Read performance consistency**: Here, Queenie just issues random reads of increasing sizes (sector-level increments) and checks whether the larger reads get higher average latencies than smaller ones. The algorithm here is `F4_incRead`.

P_7. **Read buffer capacity**: Queenie does a large sequential read first and then re-read the very first page of previous read. If the device has a buffer large enough to contain the sequential IO, then the re-read latency should be much lower than regular page read. The algorithm here is `F5_reRead`.

P_8. **Write buffer capacity**: Another probing function, `F6_conSeqW`, first erases the entire SSD and issues concurrent non-overlapping writes. When the buffer is flushed (near full), it will cause a write latency spike. The amount of data written between the latency spikes hints the size of the write buffer. The algorithm here is `F6_conSeqW`.

P_9. **Write parallelism**: While running `F6_conSeqW`, the distribution of write latencies reflects how many writes can be supported at once. As a simple example, if the later 4 of 8 concurrent writes (at the same exact time) observe almost a 2x latency compared to 4 exactly concurrent writes, then we can conclude the write buffer can absorb 4 concurrent writes at a time (*not* per second).

P_A. **Internal flush window**: After getting the write buffer size, function `F7_seqwSleep` populates the entire write buffer without triggering flush. This function then runs many iterations of filling up the buffer and also adds `sleep` (2ms to 5s in a "binary-search" manner) to figure out the minimal sleep length that completely make the latency spikes absence. This minimal length represents the internal flush window.

3.2 Probing Functions

Queenie will be open-sourced and all the pseudo-code can be found in [2]. Here, we showcase one probing function as an example. Algorithm 1 shows how `F3_strideRead` probes property P_5 (the channel/chip layout). Assuming the chunk size is already probed by previous experiment (line 4), `F3_strideRead` first securely erases the entire SSD (line 7) and then performs a full-drive sequential write with 256KB writes (line 8). It then picks a random offset aligned to

Algorithm 1: `F3_strideRead`**.** *Below is the pseudo-code to probe property P_5 (Channel/chip layout)*

```
1  $\#iterations = 5000$;
2  $\#IOs = 8/16/32/64$;
3  List⟨List⟨$lat$⟩⟩ $read\_lat$ = new List();
4  $chunk\_size = 1$ chunk;        // obtained from $P_3$
5  $size = chunk\_size$;
6  $inc\_off = chunk\_size$;
7  erase_SSD();
8  full_seqw(256K);
9  for $i = 0;\ i < \#iterations;\ i \mathrel{+}= 1$ do
10     $base\_off$ = rand_int() $\times$ $chunk\_size$;
11     $read\_lat$.add(sort(stride_read($\#IOs$, $size$,
         $base\_off$, $inc\_off$)));
12 List⟨$lat$⟩ $avg\_lat$ = average($read\_lat$);
13 $\#channels$ = analyze($avg\_lat$);
14 $read\_lat$.clear();
15 $inc\_off = chunk\_size \times \#channels$;
16 for $i = 0;\ i < \#iterations;\ i \mathrel{+}= 1$ do
17     $base\_off$ = rand_int() $\times$ $chunk\_size$;
18     $read\_lat$.add(sort(stride_read($\#IOs$, $size$,
         $base\_off$, $inc\_off$)));
19 $avg\_lat$ = average($read\_lat$);
20 $\#chips/channel$ = analyze($avg\_lat$);
```

chunks (line 10) as a base offset and sends concurrent page reads (8-64 reads) with offsets incremented by a chunk (line 11). By checking the latency distribution, to see if some reads have substantially higher latencies, `F3_strideRead` can infer the channel-level contention and thereby the #channels (line 12 and 13). After that, the offset increment can be increased to chunk size × #channels, to help probe #chips/channel using a similar approach (line 14 to 20).

3.3 Duration and Repetition

To ensure that our conjectures are highly consistent and solid, for every measurement (*e.g.*, the read latency at certain offset/size), we repeat the IOs for 5,000 to 10,000 times and uses the average, making each experiment take *1-8 hours* to finish. For experiments that require SSD erase and "refill" at the beginning of the experiment, we repeat the erase and refill ≤ 5 times (cannot be too high or the SSD will die fast).

4 Kelpie (Main Findings)

While later Section 6 and the TR [2] show all the findings, here we first highlight the 6 main findings that we consider both "interesting" and "unique." Each subsection starts with the summary of the finding.

Labeling: These SSD models will be presented with symbols composed of three parts: **protocol** ($\underline{N}$VMe, SA$\underline{T}$A,

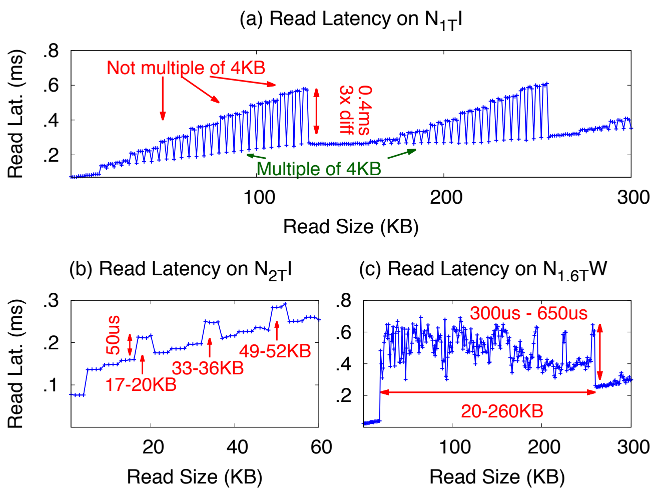

Figure 1: **Read size vs. performance.** *The figures, as explained in §4.1, show the read latencies (y-axis) observed when the read size is set to different values (x-axis). The figures show that in these three SSD models, read IOs with different sizes (un-page-aligned sizes) can result in up to 400µs or 3x higher latencies. Each point in the figure is the average latency from at least 5000 random reads with offsets aligned to a page.*

S<u>A</u>S), **<u>size</u>** (*e.g.*, $_{100G}$, $_{1T}$), and **<u>vendor code</u>** (a letter between $A..Z$). For the last item, a character represents a vendor, but the letter-to-vendor mapping is not revealed for hiding the actual vendor names. <u>As examples</u>, "$N_{1T}I$" is a 1-TB NVMe drive from vendor I, "$T_{480G}S$" is a 480GB SATA drive from vendor S, and "$A_{800G}P$" is a 800GB SAS drive from vendor P.

Disclaimers: All of our findings and conclusions are based on the observations we see as a user of the SSDs. They should be treated as *"empirically-driven, user-observed conjectures,"* as the real truth is only known by the vendors.

4.1 Read Sizes vs. Performance

Enterprise-level drives, $N_{1T}I$, $N_{2T}I$, and $N_{1.6T}W$, show higher latencies (50-400µs or up to 3x higher than the average) when the read size (a) is not a multiple of the page size or (b) lies within certain size ranges.

As SSD generally has a minimal unit of read/write called a "page," making an IO size multiple of the page size is usually recommended to avoid paying the alignment overhead [25]. Modern SSDs are mostly well-optimized to minimize this overhead to single-digit µs. However, this is not always the case even for the latest drives. One *recent* enterprise-level SSD, $N_{1T}I$, exhibits worse latencies when the read size does not fit its expectation.

Figure 1a shows that $N_{1T}I$ returns up to 400µs or 3x higher latency (y-axis) when the read sizes (x-axis) are not multiple of 4KB. To emphasize, the read offsets are page-aligned but not the sizes, and the offsets are random. With this, $N_{1T}I$ would have a downgraded performance under real industrial SSD traces, including a database workload where 65% of the reads are not multiple of 4KBs (§5.1).

Another interesting anomaly is latency spikes within certain size ranges. Figure 1b shows that drive $N_{2T}I$ responds with 50µs higher latency (20-30%) when the read sizes fall within certain size ranges (17-20KB, 33-36KB, and 49-52KB). With this knowledge, later we show that by just increasing the read size slightly to fall outside these ranges (*e.g.*, change a 20KB read to a 24KB read), one can gain a substantial improvement (§5.1). Figure 1c shows a similar anomaly in drive $N_{1.6T}W$ with a 2-3x latency (350-600µs) when read sizes fall within the 20-260KB span but nowhere else up to 1 MB (the maximum read size in our experiment).

We acknowledge that these anomalies could be a buggy firmware, "bad" chips, or many other factors. As we only have one sample of each of the models above, we cannot conclusively determine that all the models above have the same problem. Nevertheless, the fact that we (as a user) observe these anomalies, it is possible that some other deployments face a similar problem. As a future work, we will get more samples of the same model from our industry partners.

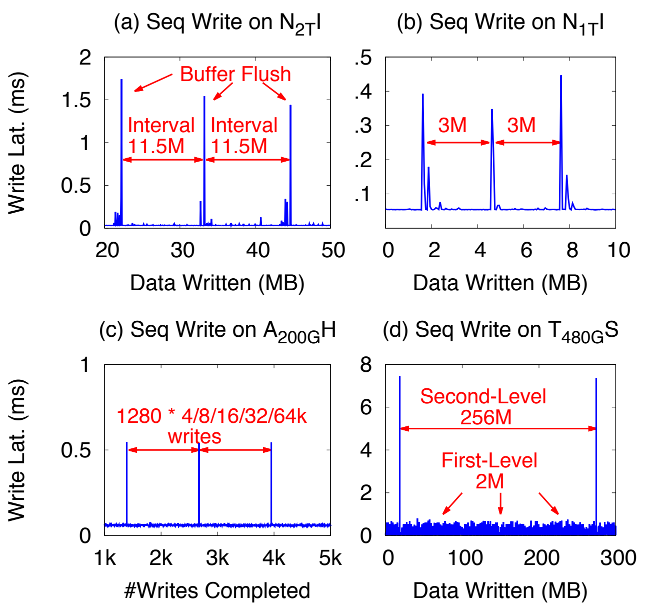

Figure 2: **Small write buffers.** *The figures, as explain in §4.2, shows the write latency (in y-axis) after a certain size of data has been written (x-axis) to the SSD. Buffered writes only take several µs, but the ms-level spikes suggest buffer flushing to the NAND cells (programming time). The distance between the two spikes hint on the write buffer size.*

4.2 Small Write Buffer

13 SSDs (mostly enterprise-level) have relatively a small write buffer ($\leq$ 64MB), while some older SSDs can employ a large buffer as high as 800MB.

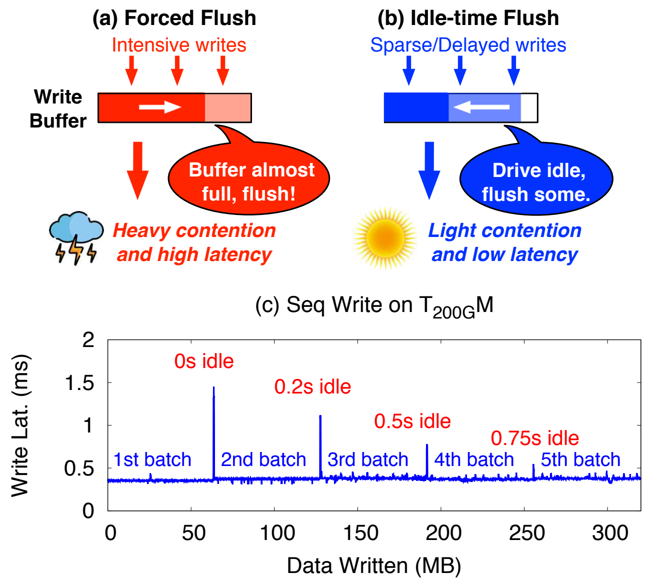

Figure 3: **Idle-time buffer flush.** *Figures (a-b), as explained in Section 4.3, illustrates the flush triggers due to heavy incoming writes or idle time, respectively. Figure (c) shows that the longer the sleep time in between the write batches, the lower latency spike the user will see.*

It is surprising to us that most SSDs, even the TBs ones, only employ a write buffer of tens of MBs. Figure 2a hints that $N_{2T}I$, a 2TB enterprise-level drive, only uses 11.5MB for the write buffer; as every 11.5MB of write results in a latency spike up to 1.5ms, which reflects the NAND programming time. Assuming the SSD is handling a decent write workload of 100 MB/s, the user would see roughly ten occurrences of latency spike. Similary, Figure 2b shows that another 1TB drive, $N_{1T}I$, can employ a "partial" write buffer of only 3MB (more detail in §6).

Another interesting observation is that write buffers on some drives are not capped by MBs but rather the number of write IOs. For instance, in Figure 2c, $A_{200G}H$ issues a flush for every 1280 writes regardless whether the size is 4/8/16/32/64k.

We also felt that 5 drives in our set employ a two-level buffering. Figure 2d shows that in $T_{480G}S$ drive, we see a write spike of almost 1ms every 2MB of write and another 7ms spike every 256MB of write, while the buffered writes only take 90µs. Unfortunately, we have no way to unearth the storage mediums behind these two-level buffering.

4.3 Idle-Time Buffer Flush

In the 14 SSDs in our benchmark, internal buffer flush is triggered during idle time, which can be exploited by making the host send sparse/delayed writes.

Internal buffer flush can be triggered during highly intensive writes (Figure 3a), which will cause a write backlog (previous section), or during idle time (Figure 3b), which can be exploited to reduce write delays by having the higher storage layers send sparse/delayed writes.

Figure 3c shows this opportunity. Here, we use $T_{200G}M$, where we already unvealed that the write buffer is 64MB. We send batches of writes, where a batch is 64MB. In between the write batches, we insert an increasing user-level sleep time that ranges between 0 to 0.75 second. In the figure, under no idle window (x=64MB), we observe a high latency spike around the boundary of the 1st and 2nd write batches, similar to those in the previous section. However as we increase the user sleep time between the subsequent batches, we see a reduced backlog. For example, as we insert 0.75s idle time between the 4th and 5th batch (x=256MB), the internal write buffer is likely being flushed, and the user writes can be absorbed to the buffer, hence low latencies.

Outcomes from the other 13 drives also show this prevalent design choice. For example, $N_{1.6T}S$ can sweep its buffer (40.25MB) in a idle window of 50ms and $T_{480G}S$ is capable of doing the same for its second-level buffer (256MB) in 5s (more in §6). We exploit this idle-time flush later in Section 5.2.

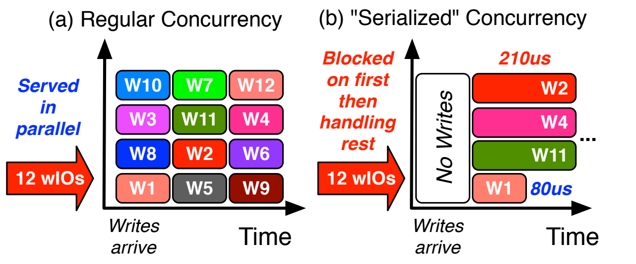

Figure 4: **"Serialized" concurrent writes.** *The figures, as explained in §4.4, show (a) an expected behavior of write parallelism with 4 concurrent completions at a time, and (b) an anomolous behavior of write parallelism where the concurrent IOs of the same batch are serialized when the write batch starts with an empty device queue.*

4.4 "Serialized" Concurrent Writes

4 out of 21 drives, $A_{960G}P_T$, $A_{960G}P_S$, $A_{1.6T}P$, and $N_{1.6T}W$, exhibit an anomolous concurrent write behavior where the concurrent IOs in a given batch are serialized when the device queue is empty of pending writes.

Regarding property P_9(write parallelism), one of Queenie functions sends continuous concurrent 32KB writes, say `w1...wz` where `1` and `z` represent the first and last write of this long-running experiment, respectively. For device $A_{960G}P_S$, Figure 4a shows that this device returns 4 write completions at a time, hence a write parallelism (P_9) of 4.

However, we notice a consistent anomolous behavior in the beginning of the experiment, and enhance Queenie to run batches of concurrent writes and *pause* in between until the

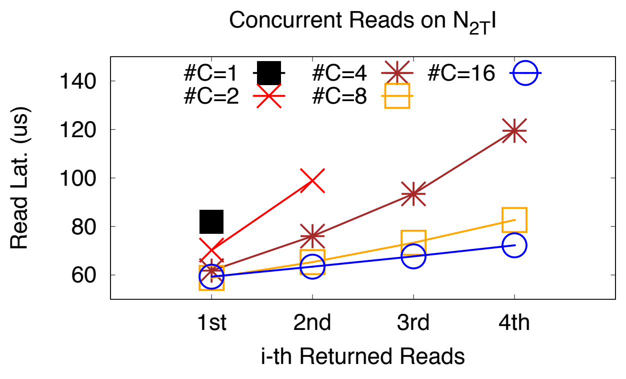

Figure 5: **Less load/concurrency, but higher latency.** *The figure as explained in §4.6 shows that in drive $N_{2T}I$, less concurrent reads lead to higher latencies than higher concurrent reads (e.g., compare the red × line for C=2 concurrent reads vs. the blue ○ line for C=16). The x-axis presents the average completion time of the i^{th} IO in every batch of concurrent reads.*

device queue has no outstanding writes. Figure 4b shows this anomolous behavior of "serialized" concurrent writes. Whenever concurrent writes are sent when the device queue has no more pending writes, we notice that the first write (`w1`) completes first after 80µs and the other concurrent writes (`w2-w4`) of the same batch complete after 210µs. Put it in other words, if users send periodic concurrent writes, these writes will be serialized in the manner as in Figure 4b.

4.5 The Disappearing Read Cache

Only 1 SSD model, $A_{800G}P$, employs an internal read cache. This disappering read cache is likely due to the increasing (a) NAND speed and (b) DRAM buffer in higher storage layers.

Internal read cache, which was an essential part of SSD years ago [25], is hardly seen in modern SSDs. As recent NVMe drives deliver low read latency (30-150µs), using internal RAM for data caching might be deemed unnecessary. Nevertheless, drives with high NAND latency can still benefit from caching. $A_{800G}P$ for example employs a 16MB read cache and reduces re-read latency from 320 to 60µs, an 80% reduction. This encouraging speed-up may motivate more SATA/SAS flash storage to adopt read caching as these older models still exhibit high read latency from 150 to 400µs.

4.6 Less Load but Higher Latency

In $N_{2T}I$ drive, lower concurrent reads unexpectedly leads to higher latencies, suggesting that some drives are built for throughput.

Usually, less concurrency/load means lower latencies. However, when we probe $N_{2T}I$'s properties, we observe an opposing behavior. In Figure 5, we perform five additional

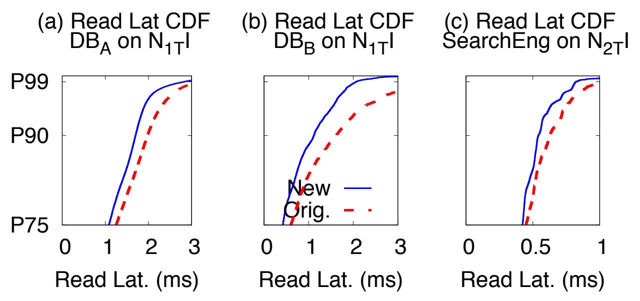

Figure 6: **Re-adjusting read sizes.** *The CDF graphs, as explained in §5.1, show a potential latency improvement when the OS/user knows about the SSD's read-size oddities. They can mitigate the sub-optimal latency (the red dashed "Orig." CDF lines) by simply altering the read sizes, resulting in lower latencies (x-axis) in high percentiles (y-axis), as shown by the "New" blue lines. DB_A, DB_B, and SearchEng represent the industrial SSD traces we use, as we run them on drive $N_{1T}I$ and $N_{2T}I$.*

experiments (the four lines and one dot), each with a different level of concurrency C from 1 to 16. An experiment sends C concurrent page random reads (as a batch), waits until the completion of the batch, and then repeats sending the same batch for thousands of times. The y-axis shows the average completion time of the i^{th} IO within every batch.

Let us start with the single ■ dot, which tells that there is only one IO (1^{st}) in the batches of "1-concurrent" IO, and the average latency is 82µs. However, in the 16-concurrent IOs experiment (the blue ○ line), we see that the average 1^{st} IO completion time (x=1) and the 4^{th} one (x=4) is only 59 and 72µs, significantly *faster* than the ■'s 82µs value. Put simply, less-loaded experiments (less concurrency in our case) has higher latencies than the more loaded ones. The root cause of this phenomenon remains a mystery. Perhaps this drive is optimized for throughput.

5 Case Studies: Using the Knowledge

This section illustrates some of the benefits of Queenie and Kelpie to storage design/policies, that is, how storage architects or users can use the extracted information to improve their storage performance. Please note that the case studies in this section are just an initial exploration. We believe that there will be more case studies to extensively use the probed information, but this is not the focus of this paper. In the case studies below, we use real industrial block-level SSD traces which represent a large company's database, search engine and cloud storage workloads.

5.1 Read Size Alignment

Let us quickly recap again the findings in Section §4.1. Here, interestingly, some datacenter SSDs exhibit higher latencies when the read sizes fall into certain ranges. As shown earlier in Figure 1, $N_{1T}I$ delivers higher latencies when read sizes are not multiple of 4KB, and $N_{2T}I$ has unexplainable 4KB-size ranges (*e.g.*, 17-20, 33-36, and 49-52KB sizes) that will result in higher latencies, and $N_{1.6T}W$'s speed drops when read sizes are in between 20-260KB.

Clearly, such behaviors are ill-suited for real user workloads. For example, in the industrial traces we use, specifically the database traces, around 65% of the read sizes are *not* multiple of 4KB but rather multiple of 512 bytes. The latency-increasing size ranges will also be an issue as even page-aligned IO sizes can fall within these peculiar ranges (*e.g.*, IO size of 24KB in $N_{1.6T}W$).

A simple straightforward rearrangment that higher storage layers can do to mitigate this issue is to adjust the read sizes. For example, for $N_{1T}I$, the OS can easily increase the read size to the next page-line boundary (*e.g.*, change a 7KB read to 8KB). Figures 6a-b show that this very simple approach can speed up two database workloads (DB$_A$ and DB$_B$), specifically, 14-31% read latency improvement at the 90th percentile, and 18-32% and 13-48% at the 95thand 99thpercentiles, respectively (the horizontal gap between the solid blue and dashed red lines).

As another example, for the read-size oddities in $N_{2T}I$, the OS can increase the read size outside the peculiar ranges (*e.g.*, change a 20KB read to 24KB). For this, we use an SSD trace from a search engine with mostly page-aligned IOs but make sure the read sizes fall outside the peculiar ranges. Figure 6c shows that the latency is improved by 14%, 16%, and 20% at the 90th, 95th, and 99th percentiles.

Other than these, open questions remain on what to do for the the $N_{1.6T}W$ case where the underperforming size range is up to 260KB. Increasing small reads to larger ones (*e.g.*, change a 40KB read to 264KB) might increase more load and cause ramifications such as more frequent read disturb.

5.2 Exploiting Write Buffer Knowledge

In Section 4.3, we show that when the internal write buffer is full, an expensive flush is triggered causing write latency spikes. What we did not show is that such an expensive flush also causes a ripple effect to *read* latencies for two reasons. First, large flushes send more writes to the NAND cells and make incoming NAND reads wait behind the longer writes due to the length of NAND programming time. Second, flushing large data will likely trigger concurrent GCs across many channels and chips at the same time, generating more read-write/erase contention compared to periodically flushing smaller data.

This begs the question: is there a way to mitigate the full buffer flush? This is where the knowledge about the internal write buffering becomes valuable (*i.e.*, the RAM buffer size and flush window). This gray-box information can be effectively used by the higher storage layers to rate-limit the incoming writes accordingly to gain performance but without delaying writes significantly.

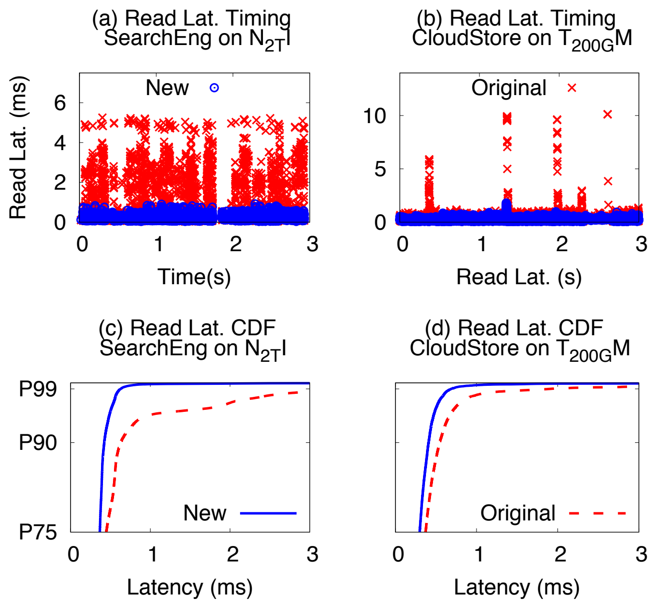

Figure 7: **Improvement from exploiting write buffer knowledge.** *The figures, as explained in §5.2, shows the read latency improvement from the rate-limiting shim layer that exploits the write buffer knowledge. The top figures show the read latencies (y-axis) across time (x-axis), where high latency spikes now disappear (no red, but more blue dots). The bottom figures show the corresponding read latency CDF of the same experiments.*

To give a quantitive sense of the improvement, we design a 4-step algorithm that can be deployed as a block-level rate-limiting shim layer. **(1)** We use Queenie to measure the internal write RAM capacity and flush window and then divide these two values to get the average "flush speed." As a concrete example, for $N_{2T}I$, we unvealed a 11.5MB buffer with 200ms flush window, giving us a flush speed of 57.5 MB/s (11.5MB/200ms). **(2)** Second, the shim layer monitors the incoming write intensity in a recent time period of configurable length. For example, a 5 MB in the last monitored period of 100ms implies a 50 MB/s intensity. **(3)** We then introduce the "flush urgency" by dividing this incoming write intensity with the flush speed; in this example, the flush urgency is 0.87 (50/57.5). This number indicates how fierce is the incoming writes with respect to the internal digest speed. **(4)** Finally, if the shim layer sees the urgency is less than 0.5, then it will let all the incoming writes within the current 100ms period to pass through into the SSD. Otherwise, the incoming writes must be slightly delayed by T ms (*e.g.*, 0.1-1ms). This T value is calculated by a rate-limiting function that incorporates the proportion of the recent read/write intensity and the flush urgency (more details in [2]).

To evaluate this rate-limiting shim layer, we run the `SearchEng` and `CloudStore` traces on $N_{2T}I$ and $T_{200G}M$, respectively. As shown in Figures 7a-b, this simple shim layer offers significant gains to read latencies. Again, in essence, this rate-limiting layer avoids a full buffer flush but rather

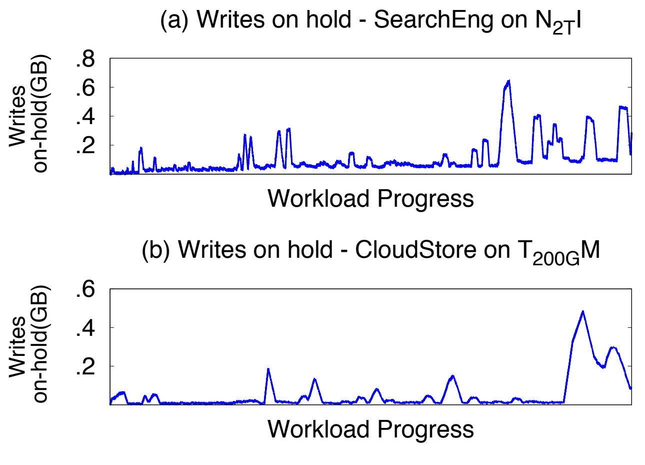

Figure 8: **On-hold data in the rate-limiting layer.** *The figures, as explain in §5.2 show the reasonable amount of writes (y-axis) held inside the rate-limiting layer across time.*

allows the underlying SSDs to flush gradually at the rate of its flush speed. Figures 7a-b show a dramatic shift from the latency spikes (red "Original" dots) to a much stable latency (blue "New" area) in both experiments. Figures 7c-d show that our rate-limiter is able to cut read latencies by 25-33%, 30-58%, and 64-82% at the 90^{th}, 95^{th}, and 99^{th} percentiles.

One reasonable concern of rate-limiting writes is how much writes the shim layer needs to hold. Figure 8 shows the timeline of the two experiments above. As shown in the y-axis, the layer only needs to hold <64 MB in every period on average, with the exception of write bursts (near the end of the timeline) which require the shim layer to hold up to 800 MB of writes for 100 seconds. To emphasize again, our algorithm here is just a simple case study. More exploration on the design space including requirements for data persistency and durability is open for future work.

6 Kelpie (All Findings)

In this last section, we show all of our findings. The Kelpie's raw data set (10 GB), which contains all the results and graphs of running Queenie on the 21 SSD models, will be made public. We now discuss each of the properties probed. Table 3.X denotes column X of Table 3.

P_1. **Page size**: 4KB is a general standard that applies to all non-SLC drives (except $A_{800G}P$) regardless of the drive capacities. In contrast, SLC drives (except $T_{200G}M$) mostly use a larger page size such as 8 or 16KB. Also, in general, recent drives tend to employ a 4KB page size. We also observed some models (*e.g.*, $N_{1.6T}S$, $N_{128G}S$ and $A_{960G}P_S$) having a dual-plane structure where two parallel pages are mapped to two adjacent planes of the same chip, hence can be simultaneously accessed in parallel. In this case, we keep the original page size but recommend to treat the two pages as a whole entity for further probing.

P_2. **Page type**: For this property, we categorize the SSDs into 3 classifications: SLC (with low pages only), MLC (low and high), and TLC (low, medium, and high). Again, low/medium/high means that the page is mapped to low/medium/high bits of the MLC/TLC cells. Based on the latencies observed, we also speculate the page-to-cell mapping and the pattern. For example, "`4L2H`" means that the first four pages of each NAND block are mapped to low pages and the next four to high pages (hence higher access latency), and the pattern `4L2H` will repeat in subsequent pages.

Table 3.P_2 shows that MLC is the most popular type in our set; MLC is known to achieve a pleasing balance between capacity, performance, and durability. For MLC models, we further find three patterns of page-to-cell mappings: `1L1H`, `4L2H`, and `4L4H`, which are more orderly patterns than non-commodity SSDs such as OpenChannel SSDs (not in our official benchmark) that exhibit a complex pattern of `6L1H2L1H2L2H...HLLH` [1] (where "..." repeats the `2L2H` pattern, as probed by a prior work [16, 4.3].

For TLC drives ($N_{2T}I$, $N_{1T}I$, and $A_{1.6T}P$), the page layout of `L`/`M`/`H` patterns are too long to be put in the table but available in our TR [2] (denoted by "`inTR`" in the Table). These drives employ a complex "composite" mapping; for example, on $N_{2T}I$, the entire 0-128KB range is mapped to `L` pages, while the 128KB-256KB range follows an `2M2H` pattern. Finally, SLC is only seen in the small-capacity, older non-NVMe SSDs (< 200 GB).

P_3. **Chunk size**: For this property, Table 3.P_3 uses KB as the unit of chunk size. For example, $N_{2T}I$'s 64KB chunk size implies that the drive maps 16 consecutive 4KB pages into a chunk. Our findings show that all SLC drives use one-page chunks. For non-SLC ones, every vendor has its own configuration. For example, vendor S uses one-page chunks, vendor P configures 4-page chunks, and vendor I organizes large chunks of 16-64 pages. Here, the chunk size can also reflects the difference in FTL mapping granularity, *e.g.*, drives with one-page chunks are more likely to use a page-level FTL mapping. As a small note, $T_{64G}S$'s chunk size is marked with a "—" as the drive has only 1 chip.

P_4. **Stripe width**: As expected, in overall, SSDs with larger capacity tend to have larger stripe width up to 128-256 of parallel chunks (chips or planes) within a stripe, showing a massive parallelism for absorbing intensive workloads. On the contrary, smaller drives usually have a stripe width of ≤64 (except $T_{200G}M$ and $A_{200G}H$). Another note is that many numbers in Table 3.P_4 are not in the power of 2, including $N_{1.6T}S$ (124), $N_{2T}I$ (186), $N_{1.6T}I$ (122), $T_{64G}I$ (20), $A_{1.6T}P$ (247), $A_{960G}P_T$ (200), $N_{1.6T}W$ (124), $A_{800G}G$ (30), and $A_{200G}H$ (253). This may indicate a couple of reasons such as parity spaces in RAIN [34, 40] or reserved/back-up chips that are not observable externally. Finally, for TLC drive $N_{1T}I$, we cannot conclusively determine the stripe width due to its complex composite mapping ("?" in Table 3).

	P_1 PgSz	P_2 PgType	P_3 ChukSz	P_4 StripeW	P_5 Layout	P_6 ReadC	P_7 RBuf	P_8 WBuf	P_9 WrPra	P_A FluWin
$N_{1.6T}S$	4K	MLC$_{\texttt{4L2H}}$	4K	124	16×8	✓	—	40.25M$_p$	8	50ms
$N_{500G}S$	4K	MLC$_{\texttt{4L2H}}$	4K	64	4×16	✓	—	2M	1	4ms
$N_{128G}S$	4K	MLC$_{\texttt{4L2H}}$	4K	32	16×2	✓	—	1M	1	2ms
$N_{2T}I$	4K	TLC$_{\texttt{inTR}}$	64K	186	12×16	✗	—	11.5M	4	200ms
$N_{1.6T}I$	4K	MLC$_{\texttt{1L1H}}$	128K	122	32×4	✓	—	11.5M	2	10ms
$N_{1T}I$	4K	TLC$_{\texttt{inTR}}$	256K	?	?	✗	—	11M$_p$	2	3ms
$N_{1.6T}W$	4K	MLC$_{\texttt{1L1H}}$	64K	124	16×8	✗	—	NB+P	4✗	—
$N_{1.6T}M$	4K	MLC$_{\texttt{1L1H}}$	128K	128	16×8	✓	—	15M	4	∞
$A_{1.6T}P$	4K	TLC$_{\texttt{inTR}}$	16K	247	16×16	✓	—	NB+P	4✗	—
$A_{960G}P_S$	4K	MLC$_{\texttt{4L4H}}$	4K	64	32×2	✓	—	NB+P	4✗	—
$A_{960G}P_T$	4K	MLC$_{\texttt{1L1H}}$	16K	200	20×10	✓	—	11.5M$_p$ | 406M	4✗	200ms | 2.5s
$A_{800G}P$	8K	MLC$_{\texttt{1L1H}}$	32K	128	16×8	✓	16M	2M | 512M	4	∞
$A_{800G}G$	4K	MLC$_{\texttt{1L1H}}$	64K	30	8×4	✓	—	3.75M	2	30ms
$A_{200G}H$	8K	SLC	8K	253	16×16	✓	—	20M | 126.5M†	1	∞
$T_{480G}S$	4K	MLC$_{\texttt{4L4H}}$	4K	16	8×2	✓	—	2M | 256M	1	10ms | 5s
$T_{200G}S$	8K	SLC	8K	64	8×8	✓	—	1M	1	35ms
$T_{128G}S$	4K	MLC$_{\texttt{4L4H}}$	4K	32	16×2	✓	—	1M	1	2ms
$T_{100G}S$	8K	SLC	8K	8	8×1	✓	—	512K	1	40ms
$T_{64G}S$	16K	SLC	—	1	1×1	✓	—	4M	2	∞
$T_{64G}I$	4K	SLC	4K	20	10×2	✓	—	10M$_p$ | 810M	1	300ms | ∞
$T_{200G}M$	4K	SLC	4K	128	8×16	✓	—	64M	1	1s

Table 3: **Summary of all findings in Kelpie.** *The entire table is fully explained in Section 6 where every P column is described in the corresponding subsection of §6. In the first column, the rows are categorized by the **protocol** (NVMe, SAS, or SATA) separated by lines. Within the lines, we then sort based on the **vendor code** (S, I, W, M, P, G, and H). We do not reveal the mapping of vendor code to the full name. The number in between denotes the drive **size** (e.g., $_{1.6T}$, $_{500G}$). For example, "$N_{1.6T}S$" is a 1.6TB NVMe drive from vendor S. Other notes: In some columns, for brevity we omit "B" (bytes), hence "K/M" means "KB/MB." "—" means not applicable. "?" implies unsuccessful probing. For other specific labels in every column (such as* `nLnH`, `inTR`, *✓, ✗, p, NB, P, †, |, ∞), please consult the corresponding subsection in §6.*

P_5. **Channel/chip layout**: Here, we found that larger drives prefer a "wide" setting – more channels but fewer chips per channel. For example, in terms of #channels × #chips/channel, $N_{1.6T}S$ has a 16×8 layout which is performant for parallelizing IOs across many channels but at the same time reduce the multi-chip bandwidth contention on each channel, and $A_{960G}P_S$ employs a 32×2 one. For other smaller drives, the setting can vary. There is a "wide" setting similar as above (the 128GB $N_{128G}S$ with 16×2) and a "deep" setting (the 500GB $N_{500G}S$ with 4×16); the deep setting is prone to many-chip bandwidth contention on every channel. We also see two drives with non-power-of-2 #channels: $T_{64G}I$ (10×2) and $A_{960G}P_T$ (20×10). For $N_{1T}I$, the layout is labeled "?" in Table 3 due to the same reason – the composite mapping issue.

One interesting anomaly that we found is that in one drive, $N_{1.6T}S$, the IOs to separate channels seem to be *contending* with each other. Upon a further investigation, we unveiled a channel "grouping," where the 16 channels on $N_{1.6T}S$ are evenly split into 4 pools, and IOs to the *same* pool will contend with each other (with an overhead of 10µs) even if the IOs go to different channels in the pool. This contradicts the traditional view of channel parallelism. The root cause remains unknown.

P_6. **Read performance consistency**: As we discussed before in Section 4.1, 3 enterprise-level SSDs ($N_{2T}I$, $N_{1T}I$ and $N_{1.6T}W$) exhibit degraded performance under certain read sizes, labeled as ✗ in Table 3.P_6. It could be a fact that applications must live with or a bug/defect inside the SSD. If it is the former, OS/applications can take mitigation ations, *e.g.*, by altering the read sizes (§5.1), but this requires a probing cost to understand if their SSDs have an inconsistency problem like this beforehand. If this is a bug/defect, then SSD vendors might want to adopt this kind of test from Queenie to their device quality tests.

P_7. **Read buffer capacity**: Our results show that read buffer is becoming extinct in modern SSDs (almost all "—" in Table 3.P_7), with the exception of one SAS drive $A_{800G}P$ which has a 16MB read buffer. This phenomenon can be attributed to the increasing speed of NAND and the larger DRAM caches in higher storage layers that make internal read cache obsolete. However, for SATA/SAS drives with higher read latency (*e.g.*, hundreds of µs), read buffer can be still beneficial (see §4.5).

P_8. **Write buffer capacity**: The first trend observed is that write-level buffering is still prevalent. But, new drives only provision a small buffer. For example, TB-level drives (*e.g.*, $N_{1.6T}S$, $N_{2T}I$, $N_{1.6T}I$, $N_{1T}I$, and $N_{1.6T}M$) use only

$\leq$64MB buffer, which could be due to many reasons. The first is lower cost and the second is that a small buffer forces frequent small flushes, which are more favorable than large flushes that can cause long blocking to both incoming reads and writes (§4.2 and §5.2) and potentially trigger large GCs.

Another trend we see is the use of 2-level buffering (§4.2), found in 5 drives labeled with a pair of first and second level sizes in Table 3.P_8. For example, $A_{800G}P$'s "2M | 512M" value implies a two-level buffer with 2MB and 512MB for the first and second levels, respectively. In one drive, $A_{200G}H$, the second-level write has two policies (marked with † in the table): flush either after a threshold of 126.5MB or 1280 IOs of writes has been exceeded, where an "IO" can be of any size.

We also found that 4 drives ($N_{1.6T}S$, $N_{1T}I$, $T_{64G}I$, and $A_{960G}P_T$) can choose to flush their write buffers partially even when they are not idle from writes. For example, under sequential writes, we see $N_{1.6T}S$ constantly flushes 5.75MB of data out of its 40.25MB full buffer capacity, $N_{1T}I$ 3MB out of 11MB, $T_{64G}I$ 4MB out of 10MB, and $A_{960G}P_T$ 7.5MB out of 11.5MB, labeled with "p" in Table 3.P_8. For SSDs, periodic partial flushes is a double-edged sword: on the one hand it tones down the blocking impact, on the other hand it can increase write amplification (*e.g.*, new ovewrites to the same LPNs just recently flushed).

Finally, we observed 3 drives ($A_{1.6T}P$, $A_{960G}P_S$, and $N_{1.6T}W$) that rarely show write latency spikes (like those shown earlier in Figure 2) even with a full write bandwidth experiment. Our assumption is that these drives perform partial flushes and are able to optimize them without blocking incoming IOs. In such an optimized design, we cannot conclusively probe their write buffer capacities and mark them with "NB + P" (non-blocking and partial) in Table 3.P_8.

P_9. **Write parallelism**: Although many SSDs have a high stripe width (P_4) to support high read parallelism, it is not the same case for write parallelism. For example, $N_{1.6T}S$, an enterprise-level drive that is able to handle 124 concurrent chunk reads can only allow 8 concurrent writes. Indeed 8 concurrent writes is the highest write parallelism that we observe, while others only reach 2 or 4 as shown in Table 3.P_9. However, we caution that read and write parallelism cannot be compared as apples to apples, mainly because read IOs fetch data from the NAND (with the absence of internal read cache) while write IOs are absorbed by the internal RAM.

As presented earlier in Section 4.4, we found anomalies where 4 drives ($A_{960G}P_T$, $A_{960G}P_S$, $A_{1.6T}P$, and $N_{1.6T}W$) exhibit "serialized" concurrent writes when the IOs are inserted to the device queue without any pending writes. These anomalies are labeled with "✗" in Table 3.P_9.

P_A. **Internal flush window**: Section 5.2 successfully demonstrates that probing the internal flush *speed* can be useful for OS/applications to rate-limit the incoming writes accordingly to improve IO performance. Flush speed is a function of buffer size (P_8) divided by the "flush window" (P_A). In our write-intensive experiments in Figure 3 of Section 4.3, the flush window is essentialy equal to how much time the OS/user should let the device being idle without seeing write latency spikes. This window value is shown in Table 3.P_A. For example, the probed window values for most of the SSDs are around 2 to 300ms.

The ∞ label in Table 3.P_A symbolizes the worst-case scenario. For 5 devices ($N_{1.6T}M$, $A_{800G}P$, $A_{200G}H$, $T_{64G}S$, and $T_{64G}I$), no matter how long an idle period is given to the device to "cool down," some writes will experience latency spikes. This basically implies that the write flush is never triggered partially in time but only when a space threshold (*e.g.*, 90% full) has been reached, and the flush will delay incoming writes.

We also found that SSDs from the same vendor have their own strategies in this context. For instance, $N_{1T}I$, $N_{1.6T}I$, and $N_{2T}I$ are three TB-level drives from the same vendor I with similar buffer sizes but use dramatically different amount of times to clean their buffers; $N_{2T}I$ chooses to do lazily within a 200ms idle window, $N_{1.6T}I$ is more agile and finish in 10ms, while $N_{1T}I$ is very progressive and dumps within 3ms.

For drives with two-level buffering, likewise, we put a pair of first and second level time windows, which shows that the drives apply different window policies for the two levels. For example, $T_{480G}S$'s "10ms | 5s" window value implies that it aggressively flushes the first level buffer in 10ms but only dumps the second level buffer lazily in 5s.

Finally, "—" in Table 3.P_A highlights that the corresponding devices ($N_{1.6T}W$, $A_{960G}P_S$, and $A_{1.6T}P$) perform the optimized non-blocking, partial flushes discussed above. Here, we cannot probe the flush window value.

7 Conclusion

Queenie has successfully probed a variety of SSDs, leading to many results and findings wrapped in Kelpie. However, we believe that there are more exciting properties to probe. For examples, in our TR [2], we summarize our failed attempts to probe the GC policies, perhaps requiring a much more sophisticated probing and statistical analysis. TLC drives seems to be more complicated FTL mapping beyond the typical static mapping. New properties such as multi-streaming might also be interesting to probe as their policies are difficult to reverse-engineered [36]. Finally, not just the probing phase, but also the question of how the OS/applications can exploit all of this unveiled internal knowledge needs to be continuously answered.

In conclusion, as mentioned in the introduction, we strongly believe that more "Newt"s (more investigating and investigators) are needed to solve the "crimes" of the complexities, intricacies, and nuances of modern state-of-the-art SSDs. We hope that Queenie and Kelpie will spur more exciting future research in this space.

. .

*This page is **intentionally left blank** so that the 2-page references start from an odd-numbered page. This way, with duplex printing, the reviewers can easily separate the 1-sheet reference section from the main paper for a more comfortable reading.*

References

[1] Open-Channel Solid State Drives Specification. http://lightnvm.io/docs/OCSSD-2_0-20180129.pdf.

[2] Queenie and Kelpie - Complementary Technical Report. https://bit.ly/2ngxf7r.

[3] RAIL: Predictable, Low Tail Latency for NVMe Storage. https://platformlab.stanford.edu/Presentations/Litz.pdf, 2018.

[4] SOLID STATE DRIVE (SSD) MARKET - GROWTH, TRENDS, AND FORECAST (2019 - 2024). https://www.mordorintelligence.com/industry-reports/solid-state-drive-market, 2019.

[5] Abutalib Aghayev and Peter Desnoyers. Skylight-A Window on Shingled Disk Operation. In *Proceedings of the 13th USENIX Symposium on File and Storage Technologies (FAST)*, 2015.

[6] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2008.

[7] Matias Bjorling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[8] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST)*, 2010.

[9] Simona Boboila and Peter Desnoyers. Performance Models of Flash-based Solid-State Drives for Real Workloads. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[10] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. On the Performance Variation in Modern Storage Stacks. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[11] Feng Chen, Rubao Lee, and Xiaodong Zhang. 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)*, 2011.

[12] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 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)*, 2015.

[13] Timothy E. Denehy, John Bent, Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, 2004.

[14] Peter Desnoyers. Analytic Models of SSD Write Performance. In *ACM Transactions on Storage (TOS)*, 2014.

[15] Laura M. Grupp, John D. Davis, and Steven Swanson. The Harey Tortoise: Managing Heterogeneous Write Performance in SSDs. In *Proceedings of the 2013 USENIX Annual Technical Conference (ATC)*, 2013.

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

[17] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Symposium on File and Storage Technologies (FAST)*, 2016.

[18] Benny Van Houdt. A Mean Field Model for a Class of Garbage Collection Algorithms in Flash-based Solid State Drives. In *Proceedings of the 2013 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013.

[19] Xiao-Yu Hu, Robert Haas, and Eleftheriou Evangelos. Container Marking: Combining Data Placement, Garbage Collection and Wear Leveling for Flash. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011.

[20] H. Howie Huang, Shan Li, Alexander S. Szalay, and Andreas Terzis. Performance Modeling and Analysis of Flash-based Storage Devices. In *Proceedings of the 27th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2011.

[21] Myoungsoo Jung and Mahmut Kandemir. Revisiting Widely Held SSD Expectations and Rethinking System-Level Implications. In *Proceedings of the 2013 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2013.

[22] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, , and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2014.

[23] Jae-Hong Kim, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. A Methodology for Extracting Performance Parameters in Solid State Disks (SSDs). In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2009.

[24] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, 2019.

[25] Jaehong Kim, Sangwon Seo, Dawoon Jung, Jin-Soo Kim, and Jaehyuk Huh. Parameter-Aware I/O Management for

Solid State Disks (SSDs). In *IEEE Transactions on Computers (TC)*, 2011.

[26] Jihun Kim, Joonsung Kim, Pyeongsu Park, Jong Kim, and Jangwoo Kim. SSD Performance Modeling Using Bottleneck Analysis. *IEEE Comput. Archit. Lett.*, 17(1):80–83, January 2018.

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

[28] Pradeep Kumar and H. Howie Huang. Falcon: Scaling IO Performance in Multi-SSD Volumes. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, 2017.

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

[30] Shan Li and H. Howie Huang. Black-Box Performance Modeling for Solid-State Drives. In *Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2010.

[31] Yongkun Li, Patrick P. C. Lee, and John C. S. Lui. Stochastic Modeling of Large-Scale Solid-State Storage Systems: Analysis, Design Tradeoffs and Optimization. In *Proceedings of the 2011 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2011.

[32] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-Defined Flash for Web-Scale Internet Storage System. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[33] Torben Kling Petersen and John Bent. Hybrid Flash Arrays for HPC Storage Systems: An Alternative to Burst Buffers. In *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[34] Sergey Platonov. RAIN: Reinvention of RAID for the World of NVMe. In *Flash Memory Summit*, 2018.

[35] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST)*, 2002.

[36] Arash Tavakkol, Juan Gomez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *Proceedings of the 16th USENIX Symposium on File and Storage Technologies (FAST)*, 2018.

[37] Brent Welch and Geoffrey Noer. Optimizing a Hybrid SSD/HDD HPC Storage System Based on File Size Distributions. In *Proceedings of the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST)*, 2013.

[38] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 1995.

[39] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. Garbage Collection Aware Cache Management with Improved Performance for Flash-based SSDs. In *Proceedings of the 30th International Conference on Supercomputing (ICS)*, 2016.

[40] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. In *Proceedings of the 15th USENIX Symposium on File and Storage Technologies (FAST)*, 2017.

[41] Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic Measurement of Memory Hierarchy Parameters. In *Proceedings of the 2005 ACM Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*, 2005.

[42] Jianquan Zhang, Dan Feng, Jianlin Gao, Wei Tong, Jingning Liu, Yu Hua, Yang Gao, Caihua Fang, Wen Xia, Feiling Fu, and Yaqing Li. Application-Aware and Software-Defined SSD Scheme for Tencent Large-Scale Storage System. In *Proceedings of 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2016.

[43] Aviad Zuck, Philipp Guhring, Tao Zhang, Donald E. Porter, and Dan Tsafrir. Why and How to Increase SSD Performance Transparency. In *The 17th Workshop on Hot Topics in Operating Systems (HotOS XVII)*, 2019.