Appendices

Queenie — Probing Experiments and Pseudo Code

- Nanqinqin Li

A Probing Experiments

Our methodology is based on patterned workloads and statistical analysis (e.g., sort, average, and percentile data). Basically, patterned workloads refer to series of I/Os that follows some pre-designed patterns. The patterned workloads will be replayed on the target SSDs, and the latency of each individual I/O will be precisely measured as output. Then, we manually apply some statistical methods to analyze the output (i.e., latency distribution) and speculate what kinds of SSD internal designs would result in this specific output shape. The idea of using patterned workloads is that SSDs adopt some repeatable design patterns, and when some parts of our patterned workloads are consistent with those lie within SSDs, we can see some expected output.

We designed seven core experiments dedicated for uncovering different SSD properties (table 1), and table 2 is a list of important utility methods used in these experiments. In algs. 1-5, the pseudo-code for ***Push Read*** (`F1_pushRead`) experiment [1], ***Range Read*** (`F2_rangeRead`) experiment, ***Stride Read*** (`F3_strideRead`) experiment [1], ***Increment Read*** (`F4_incRead`) experiment, and ***Re-Read First Page*** (`F5_reRead`) experiment is listed. These five experiments are capable of uncovering SSD read-related features such as page size, stripe width, and read buffer size. In alg. 6, the pseudo-code for ***Concurrent Sequential Write*** (`F6_conSeqW`) experiment and ***Seq Write with Sleep Injection*** (`F7_seqwSleep`) experiment is listed. These two experiments are effective in extracting write-related properties such as the write buffer capacity.

Experiment	Properties	Threads
`F1_pushRead`	P_1, P_3	single
`F2_rangeRead`	P_2	single
`F3_strideRead`	P_4, P_5	multiple
`F4_incRead`	P_6	single
`F5_reRead`	P_7	single
`F6_conSeqW`	P_8, P_9	multiple
`F7_seqwSleep`	P_{10}	single

Table 1: **Target properties of each probing experiment.**

For read experiments, the SSDs first need to be erased and then be refilled (a full sequential write). This erase-and-refill only needs to be set up once for all read experiments. For write experiments, it is not necessary to erase the whole drive every time before running experiments as long as no overwriting is guaranteed, that is, always make sure that data will be written to empty pages. We recommend to finish all read experiments before doing write ones to minimize the erase count. In our entire experiments and analysis, no drive has been erased more than five times.

Utility methods definitions
- `erase_SSD()`: issues a secure erase to the target SSD.
- `full_seqw`(*bs*): issues a full sequential write with block size *bs* to the target SSD.
- `rand_offset`(*A*): generates a random offset aligned to *A*.
- `read`(*S*, *O*): issues a read request of size *S* at offset *O*.
- `write`(*S*, *O*): issues a write request of size *S* at offset *O*.
- `stride_read`(*J*, *S*, *O*, *St*): issues *J* number of concurrent read requests with size *S* and with a stride *St* in offset starting at *O*. Returns a list of latency.
- `stride_write`(*J*, *S*, *O*, *St*): issues *J* number of concurrent write requests with size *S* and with a stride *St* in offset starting at *O*. Returns a list of latency.
- `sleep`(*T*): sleeps for *T* amount of time.
- `avg`(*L*): calculates the average for the latency list *L*.
- `sort`(*L*): sorts the latency list *L*.
- `analyze`(*L*): the automated analysis that generates the probed properties.

Table 2: **Utility methods used in experiments.**

A.1 Push Read

`F1_pushRead` is inspired by previous work [1] and has been revised for more properties. In this paper, `F1_pushRead` can help reveal P_1 and P_3.

P_1. **Page size:** For P_1, the idea is to use sectors, the smallest addressable units, to probe the borders between pages in order to identify the page size. As shown in alg. 1, first, we choose $num = 2$. Then, for every iteration (the inner for loop), we randomly generate a 256KB-aligned offset. We assume that this random offset is aligned to the beginning of a page unless the actual page size is larger than 256KB or it is not a multiple of 4KB, which is quite unlikely for SSDs. Thus, as the additional offset a increases (the outer for loop), these two sectors will be pushed further away, and when they sit on the page borders (in other words, these two sectors get separated) the SSD would have to read two pages instead of one since page is the atomic read/write unit. As an illustration, say that $a = 7$ and the actual page size is 8 sectors (4KB), the border would split these two sectors into two pages, causing two pages being read. So, as the additional offset keeps increasing, we can expect periodic latency

Algorithm 1: Pseudo-code for `F1_pushRead`

```
Input: $num$, $\#iterations$
erase_SSD();
full_seqw(256 KB);
$sec = 512$ B;
$size = num * sec$;
$avg\_lat = []$;
for $a = 0$; $a \leq 256$ KB; $a$ += $sec$ do
    $lat = []$;
    for $i = 0$; $i < \#iterations$; $i$ += 1 do
        $offset$ = rand_offset(256 KB);
        $lat$.add(read($size$, $offset + a$));
    $avg\_lat$.add(avg($lat$));
$page\_size$ = analyze($avg\_lat$);
```

spikes (reading 2 pages vs reading 1 page), and the distance between two back-to- back spikes is the page size.

Also, we can use other numbers for num to verify the conclusion. For example, if $num = 4$, we would be seeing periodic three-spike groups showing up, and the distance between the first/second/third spikes of two back-to-back spike groups should also be the page size.

P_3. **Chunk size:** As for the periodic latency spikes, there could two different situations of reading these two pages. The first one is that these two pages are on the same chip and get served on a FCFS basis (First-come-first-serve, in our context, this refers to a blocking manner where the the second one has to wait the first to finish). Thus, we expect to see almost-double latency spikes compared to reading one page. The other one is the two pages are on another chip which means much lower spike since they can go in parallel. For the first case, we further derive that the chunk size, P_3, is only one page. For the second case, we need to configure `F1_pushRead` in order to probe the chunk size.

We only need to set $sec = 1$ page in line 3 and choose a larger range for additional offset a (i.e., 1M). Similarly, also take $num = 2$ as an example, as the additional offset a is pushing these two pages away from the starting position, when they sit on the borders of chunks, these two separated pages can be served by two chips in parallel, causing a lower latency. Thus, the distance between two back-to-back latency dips is the chunk size.

A.2 Range Read

P_2. **Page type:** `F2_rangeRead` (alg. 2) simply probes a certain LPN/LBA range (from $start$ to end) for a given SSD and divides the latencies into categories, low pages with lower latency and high pages with higher latency. If we get two or more categories, we believe that the target SSD is multilevel- celled (MLC) and SLC for getting one category. To be more specific, in this paper, we use SLC to represent 1-bit cell type (one category of latencies), MLC for 2-bit (two categories of latencies), and TLC for 3-bit (three categories of latencies), as this representation is widely used in the SSD industry. Meanwhile, the probing results would also show repeatable low-high page patterns, as being referred as page layout in this paper.

Algorithm 2: Pseudo-code for `F2_rangeRead`

```
Input: $start$, $end$, $\#iterations$
erase_SSD();
full_seqw(256 KB);
$size = 1$ page;
$offset$ = rand_offset($size$);
$avg\_lat = []$;
for $a = end$; $a \geq start$; $a$ −= $size$ do
    $lat = []$;
    for $i = 0$; $i < \#iterations$; $i$ += 1 do
        $lat$.add(read($size$, $offset + a$));
    $avg\_lat$.add(avg($lat$));
($page\_type$, $page\_layout$) = analyze($avg\_lat$);
```

Algorithm 3: Pseudo-code for `F3_strideRead`

```
Input: $\#IOs$, $max$, $\#iterations$
erase_SSD();
full_seqw(256 KB);
$size = 1$ chunk;
$avg\_lat = [[]]$;
for $s = 0$; $s \leq max$; $s$ += 1 do
    $lat = [[]]$;
    for $i = 0$; $i < \#iterations$; $i$ += 1 do
        $offset$ = rand_offset(256 KB);
        $lat$.add(sort(stride_read($\#IOs$, $size$,
         $offset$, $s$)));
    $avg\_lat$.add(avg($lat$));
($stripe\_width$, $\#channels$, $\#chips$) =
  analyze($avg\_lat$);
```

We recommend to check whether the target SSD adopts read caching or read-ahead mechanism before conducting `F2_rangeRead`. If it does, the `F2_rangeRead` workload must be carefully randomized to minimize the noises. Meanwhile, one trick, as shown in line 6, can be deployed to get reliable latency results: the probing order is from largest LPN to lowest. Another trick is that, we can also use percentile data to do the result analysis (e.g., the 90th percentile of latency list lat) if the average cannot be trusted.

A.3 Stride Read

`F3_strideRead` (alg. 3) is built upon ideas from previous work [1] with necessary enhancements to cover more properties. It is designed to uncover P_4 and P_5. The key of

`F3_strideRead` is the stride_read method that issues multiple concurrent read requests in order to probe the internal parallelism of SSDs. The last parameter St of stride_read is called the offset stride. For example, if stride_read issues 4 reads with base offset O and stride St, the actual offsets of these 4 reads are O, $O + St$, $O + 2 \times St$, and $O + 3 \times St$.

Please note that it is impossible to determine the actual processing order of those concurrent IOs. Instead, we use the return order. After getting the latency list of a given concurrent IO batch, we sort the latency from smallest to largest and consider the lowest latency as the first one processed by the SSD.

P_4. **Stripe width:** As for P_4, we set $size = 1$ chunk, $\#IOs = 8/16/32$, and let the offset stride s ranges from 0 to a reasonable number (i.e., 1024). As the stride increases, `F3_strideRead` is able to cover different cases of contention of these concurrent read requests. For example, when s equals to the stripe width of the target SSD, all the read requests will be sent to the same chip, causing the highest degree of contention (all reads will be served in a FCFS manner), while using half a stripe width as s would result in two groups of contending reads (each group has $\#IOs/2$ reads). Ideally, the latency spike of the second case should be about half of the first case. Thus, we can expect multiple levels of spikes, and the more $\#IOs$ the more levels. Based on this, the distance between two back-to-back highest spikes is the stripe width.

P_5. **Channel/chip layout:** As for P_5, here we describe a setting for `F3_strideRead` different from the one shown in the paper but also works. To uncover $\#channels$, we set $\#IOs = 2$. Similarly, when s equals to the stripe width, these two reads contends with each other inside a chip (chip-level contention). However, when s equals to $\#channels$, these two reads will be sent to the same channel but different chips, causing channel-level contention, which would has a spike lower than chip-level contention. Thus, the distance between two back-to-back channel-level spikes is the $\#channels$.

A.4 *Increment Read*

P_6. **Read performance consistency:** `F4_incRead` (alg. 4) thoroughly probes the latency of different read sizes, incrementing from a small size to a large size. When we use an increment unit inc smaller than a page (i.e., a sector), we can verify the conclusion of P_1 and also get a sense of the page alignment overhead as we are serving SSDs with non-page-aligned read workload. Also, we can observe the performance of SSDs handling large read requests such as 256KB and 512KB. Ideally, we would hope SSDs perform consistently as the size varies, that is, in general, large read requests should have higher latency than small ones.

Please also note that in `F4_incRead` we align the offsets of each read to 256KB and make the size sector-aligned. One can align the offsets to 512B and fix the read size to page-aligned to check whether the SSD handles these kind of situation well.

Algorithm 4: Pseudo-code for `F4_incRead`

```
Input: $max$, $\#iterations$
erase_SSD();
full_seqw(256 KB);
$inc = 512\,\text{B}$;
$avg\_lat = []$;
for $size = inc$; $size \le max$; $size$ += $inc$ do
    $lat = []$;
    for $i = 0$; $i < \#iterations$; $i$ += 1 do
        $offset$ = rand_offset(256 KB);
        $lat$.add(read($size$, $offset$));
    $avg\_lat$.add(avg($lat$));
$read\_consistensy$ = analyze($avg\_lat$);
```

Algorithm 5: Pseudo-code for `F5_reRead`

```
Input: $max$, $\#iterations$
erase_SSD();
full_seqw(256 KB);
$inc = 512\,\text{B}$;
$avg\_lat = []$;
for $size = inc$; $size \le max$; $size$ += $inc$ do
    $lat = []$;
    for $i = 0$; $i < \#iterations$; $i$ += 1 do
        $offset$ = rand_offset(256 KB);
        read($size$, $offset$);
        $lat$.add(read(1 page, $offset$));
    $avg\_lat$.add(avg($lat$));
$read\_buffer$ = analyze($avg\_lat$);
```

A.5 *Re-Read First Page*

P_7. **Read buffer capacity:** `F5_reRead` (alg. 5) reads a large amount of sequential data and then immediately re-read the first page of that data. The idea is that if the data size is less than or equal to the target SSD's read buffer capacity, the first page of data should still be cached. Otherwise, the first page should be cast out. Thus, the average latency of reading the first page as the large read increases can tell us when do we run out of the read buffer.

A.6 *Concurrent Sequential Write* and *Seq Write with Sleep Injection*

`F6_conSeqW` and `F7_seqwSleep` (alg. 6) basically have the same functionality except that `F7_seqwSleep` injects a sleep operation between each write operation. Similar to `F3_strideRead`, the essential part of these two experiments

Algorithm 6: Pseudo-code for `F6_conSeqW` and `F7_seqwSleep`

```
Input: $\#IOs$, $bs$, $base\_off$, $\#iterations$, $s\_time$
1 erase_SSD();
2 $all\_lat = []$;
3 for $i = 0$; $i < \#iterations$; $i \mathrel{+}= 1$ do
4     $all\_lat$.add.(sort(stride_write($\#IOs$, $bs$, $base\_off$, $bs$)));
5     $base\_off \mathrel{+}= \#IOs * bs$;
6     sleep($s\_time$);
7 $(write\_buffer, write\_para, flush\_window) =$ analyze($all\_lat$);
```

is the stride_write method. Just like stride_read, this method issues multiple concurrent write requests with an offset stride separating them. In `F6_conSeqW` and `F7_seqwSleep`, the offset stride is fixed to be the same as the write size to ensure a sequential workload, while this setting also rules out the possibility of overwriting.

P_8. **Write buffer capacity:** To probe P_8, we set $\#IOs = 1$, pick a reasonable bs (i.e., 64KB), and let `F6_conSeqW` keep writing a large amount of sequential data (i.e., $\#iterations = 10^6$). If the target SSD adopts write buffer to absorb the write data, we should be able to see periodic write latency spikes as expensive flush operations would occur when the buffer gets filled up. The intervals (total data written) between these spikes should be similar, indicating the write buffer size. We may also use different $\#IOs$ settings to verify the buffer size. Similarly, instead of observing the write latency, we can keep performing background reads while conducting sequential write, and the background read workload should also show up periodic latency spikes as the flush operations would also block read [2].

For some SSD models, `F6_conSeqW` may not work as expected. In this case, we have the following remedies:

- **Block size configuration.** Originally, we conduct 7 sets of `F6_conSeqW` on each drive with the block size bs to be 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, and 256KB, respectively. We also recommend using prime numbers such as 3KB, 7KB, 13KB, 31KB, 61KB, 127KB, and 251KB.

- **Random Write.** Since the workload is sequentially arranged, pre-processing may involve. Random write injects randomness which may help avoid some unexpected situations such as pre-processing that prevent us from observing flush. Please notice that we should carefully arrange the random workload in a way that overwriting can be avoided.

P_9. **Write parallelism:** To probe P_9, we need to set $\#IOs = 8/16/32$ in order to observe the latency distribution. As IOs served in parallel should have similar latencies, we can divide these concurrent IOs into groups based on the latency level. For example, if we issue 8 concurrent 32K writes to the target SSD and we get two latency groups, (111, 114, 120, 127µs) and (160, 163, 169, and 179µs). This case highly suggests that the target SSD can serve 4 writes at a time. Another way to interpret this latency distribution is that, if these writes were serve in a FCFS manner, the largest latency should be something like 700µs (a little bit lower that 111µs *8). However, the target SSD responded with a 179µs of completion time, indicating a 4x speed boost compared to FCFS, which also means 4 writes can go in parallel. P_9 requires a close inspection on the latency distribution of each individual IO batch.

P_{10}. **Internal flush window:** To probe this, an intuitive way is to inject sleep operation right before triggering flush. For instance, given an SSD with 64MB of write buffer size, we keep writing data until the buffer is almost full, inject a sleep, and then resume writing to see if we get a flush spike. However, this approach is extremely difficult to control under some circumstances as we are focusing on each individual flush occurrence. If we wrongly pinpoint the timing of one flush, we may miss all of them, which would lead to biased observations.

To be more effective and reliable, `F7_seqwSleep` carefully manipulates the intensity of the sequential write workload by injecting sleep operations between each write operation. We believe that the idle time granted by sleep gives the SSDs chances to perform background flush. Intuitively, the longer the idle time, the more background flush occurrences, meaning that the possibility of triggering a full flush by filling up the write buffer gets lower. To get an accurate time window, we need to run multiple times of `F7_seqwSleep`. We can gradually increase the duration of the sleep operations (e.g., from 10µs to 10ms) until we see no flush spikes. Then, for that given sleep-injected sequential workload, the average time to write the amount of data same as the buffer size is the internal flush window P_{10}. As an illustration, given an SSD with 64MB of write buffer, when conducting `F7_seqwSleep` without sleep, it takes around 400ms to finish writing 64MB of data with periodic flush spikes showing up. As we gradually increase the sleep duration, the time needed to finish 64MB of write data will also increase. When that time reaches 5s, the flush spikes completely disappear. Then, we conclude that the internal flush window for this drive is 5s.

Also, to find the actual window more efficient, one can apply binary search to help precisely tune the sleep duration.

References

[1] 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.

[2] 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.