Degraded mode-benefited I/O scheduling to ensure I/O responsiveness in RAID-enabled SSDs

Zhibing Sha, Jun Li, Zhigang Cai, Min Huang, Jianwei Liao, François Trahay

Degraded Mode-benefited I/O Scheduling to Ensure I/O Responsiveness in RAID-enabled SSDs

ZHIBING SHA, JUN LI, ZHIGANG CAI, MIN HUANG, and JIANWEI LIAO, Southwest University of China, China

FRANCOIS TRAHAY, Télécom SudParis, France

RAID-enabled SSDs commonly have unbalanced I/O workloads on their components (e.g. SSD channels), as the data/parity chunks in the same stripe may have varied access frequency, which greatly impacts I/O responsiveness. This paper proposes a I/O scheduling scheme by resorting to the degraded read mode and the read-modify-write mode, to reduce the long-tail latency of I/O requests in RAID-enabled SSDs. The basic idea is to avoid scheduling read or update requests to the heavily congested but targeted RAID components. Such requests are satisfied by accessing other relevant RAID components by certain XOR computations (we call *the degraded modes*). Specially, we build a queuing overhead assessment model on the top of factors of data redundancy and the current blocked I/O traffics on SSD channels, to precisely dispatch incoming I/O requests to be fulfilled with the degraded mode or not. The trace-driven experiments illustrate that the proposed scheme can reduce the long-tail latency of read requests by 23.1% on average at the 99.99th percentile, in contrast to state-of-the-art scheduling methods.

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Reliability**;

Additional Key Words and Phrases: SSDs, RAID-5, Degraded Mode, Long-tail Latency, Modeling, Reliability

ACM Reference Format:
Zhibing Sha, Jun Li, Zhigang Cai, Min Huang, Jianwei Liao, and Francois Trahay. 2022. Degraded Mode-benefited I/O Scheduling to Ensure I/O Responsiveness in RAID-enabled SSDs. 1, 1 (March 2022), 22 pages. https://doi.org/10.1145/3522755

1 INTRODUCTION

The NAND-based solid-state drivers (SSDs) have been widely employed in consumer devices and high performance computing platforms, thanks to their features of random access, lower power consumption, and collectively massive parallelism [1, 2]. Though early SSDs were prohibitively expensive, the emergence of Multi-Level Cell (MLC), Triple-Level Cell (TLC) , and even Quad-Level cell (QLC) technologies has significantly driven down the per-unit price of SSDs with keeping multiple bits in a NAND cell [3–5]. Such high density SSD devices, however, are severely impacted by read/write disturb, data retention and low disturbance endurance that directly increase the raw bit error rate (RBER) [3, 6]. Therefore, efficiently dealing with such noises to ensure the reliability of flash memory-based SSDs becomes a challenging issue [2, 7, 8]. It is true that advanced Error Correction Code (ECC) schemes, such as Low Density Parity Check Code (LDPC) can easily cover RBERs at the cost of the latencies from read retries [2, 9], but they cannot correct chip/channel-level failures in SSDs[10, 11].

Device-level redundancy is the first line of defense to confront storage hardware failures. The approach of parity-based redundant array of inexpensive disks (e.g. *RAID-3*/*RAID-5*) is utilized inside SSDs to cope with chip/channel level

Corresponding author: J. Liao, liaojianwei@il.is.s.u-tokyo.ac.jp. He works for College of Computer and Information Science, Southwest University of China, and State Key Lab. for Novel Software Technology, Nanjing University, P.R. China.
Authors' addresses: Zhibing Sha; Jun Li; Zhigang Cai; Min Huang; Jianwei Liao, Southwest University of China, Chongqing, China, 400715; Francois Trahay, Télécom SudParis, Evry, France, 91011.

2022. Manuscript submitted to ACM

failures, to offer high reliable end data service in compact SSDs [3, 6, 12]. An SSD consists of multiple channels with each one having one or multiple chips, and each channel can work in parallel just like an independent disk does. That is to say, the multi-channel structure offers opportunity to implement RAID into a single SSD to form a channel-RAID [13]. Generally, a (data/parity) chunk is normally referred to a page in RAID-enabled SSDs, and all chunks belonging to the same data stripe will be distributed across all associated chips/channels. In case either of chips/channels is broken, we can restore all the lost data/parity chunks by reading other relevant RAID components accompanying with certain XOR computations. In this paper, we make use of the channel-level RAID implementation as the default case for illustration.

On the one hand, due to the natures of `out-of-place update` and `round-robin parity placement` at the flash level of SSDs, conventional RAID implementations may result in uneven data workloads across all channels of SSDs [6]. In other words, certain SSD channels maintain hot read/write data chunks, so they must service more I/O requests and then endure more garbage collections (GCs). Consequently, I/O requests that target at hot channels are delayed, which exacerbates long-tail latency of I/O requests and then greatly affects I/O responsiveness in RAID-enabled SSDs [14].

On the other side, users anticipate fast and stable latencies [15], but SSDs do not always deliver the expected real-time I/O services and thus some even suggest that flash storage "*may not save the world*" because of the tail latency problem [16]. The main reason for tail latency in RAID-enabled SSDs is that different RAID components (e.g. SSD channels) have uneven busyness on I/Os and GCs while running user applications [17]. Especially with respect to the issue of mitigating the negative effects of garbage collection that is the heaviest operation in SSDs, I/O requests on the GC target channels will be fulfilled by reading the data on other channels of the same stripe with certain XOR computations [14, 18, 19].

In order to essentially balance I/O access workloads over all channels of RAID-enabled SSDs and then ensure I/O responsiveness from the source, this paper proposes degraded mode-benefited I/O scheduling. Specifically, it intends to satisfy a part of I/O requests on the most congested SSD channels through accessing other relevant SSD channels (termed as ***the degraded mode***). In brief, this paper makes following three contributions:

- We introduce *degraded mode-based I/O scheduling* that balances I/O access workloads over all channels in RAID-enabled SSDs. To speedup reading/updating the data chunk on a congested channel that may have intensive I/O requests or GC operations, a degraded mode operation is issued to fetch the remainder (data/parity) chunks of the same stripe for regenerating the expected chunk data with certain XOR computations.
- We build a queuing overhead assessment model that takes the factors of data redundancy and blocked I/O traffics on SSD channels into account. Then, this model can help deciding whether the current I/O request should be fulfilled with the degraded mode or not during I/O scheduling.
- We conduct preliminary evaluation on several block I/O traces of real-world applications. As measurements indicate, our proposal can afford a better performance improvement on the metrics of overall I/O response time, I/O long-tail latency and I/O workload balance.

The remainder of paper is organized as follows: Section 2 depicts background knowledge, related work and motivations. The approach of degraded mode-based I/O dispatching in RAID-enabled SSDs is specifically presented in Section 3. Section 4 describes the evaluation experiments and relevant discussions. Finally, the paper is concluded in Section 5.

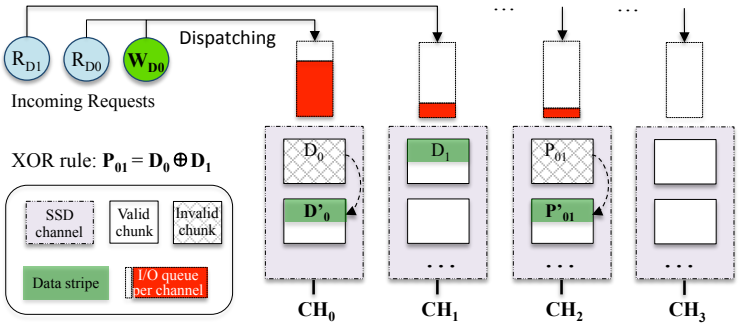


Fig. 1. I/O processing in channel-level RAID-enabled SSDs (for simplicity of illustration, each stripe includes two data chunks and one parity chunk).

2 RELATED WORK AND MOTIVATION

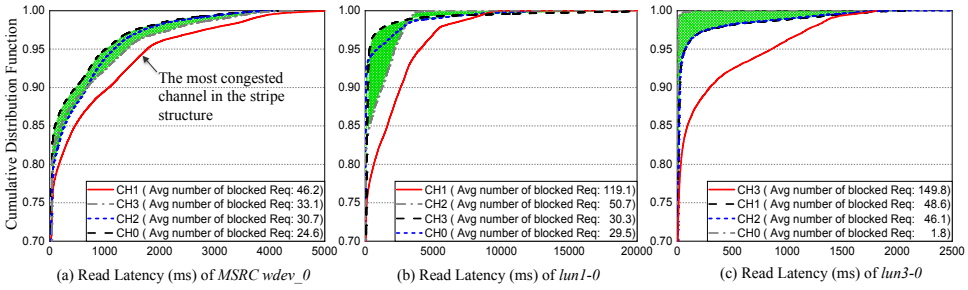
2.1 Background and Related Work

As discussed, SSD devices have a functionality of ECC to guarantee data integrity when reading a data page [2, 9]. If errors are detected and the number or span of errors is beyond the ECC capability, then the read operation is deemed a failure and the SSD device is notified. In such cases, RAID is used to regenerate the data and write a new copy onto a free data page of SSD. As one of standard RAID levels, *RAID-5* consists of block-level striping with distributed parity, and has advantages in load-balancing and I/O parallelism, so it has been commonly applied in SSDs targeting at either chip-level [6] or channel-level, for the purpose of reliability [20]. Figure 1 shows our example of I/O processing in channel-level RAID implementation. For simplicity of illustration, we use four channels (CH_0-CH_3) and each data stripe has two data chunks and one parity chunk.

According to the nature of *RAID-5*, whenever a data chunk is updated, both the original data chunk and the corresponding parity chunk in the same stripe are marked as invalid first, and then the new data chunk and the parity chunk are flushed onto the same channels. As seen in Figure 1, the stripe (D_0, D_1, P_{01}) is updated to (D'_0, D_1, P'_{01}), after completing the request of W_{D0} that intends to modify the data chunk of D_0 saved on CH_0. Note that the updated data chunks or parity chunk should be placed on the same channel in RAID-enabled SSDs, by keeping the stripe structure. Specially, each update request has to read the old chunk, and then merge it with the coming new data to form the new data chunk, which is also called *Read-Modify-Write* [12].

The SSD channel having hot updated data chunks may induce a large number of read and write requests, as well as more garbage collections. As a result, the problem of long-tail latency on such channels becomes even worse. In order to reduce the number of parity writes, Kim et al. [12] proposed a method to dynamically reconstruct stripes with a flexible size, for placing the data evenly on all RAID components (i.e. SSD chips in their context). But this method brings about complicated data structures of mapping table, and cannot balance I/O workloads over all RAID components from the source.

Furthermore, the degraded mode-like optimization strategies are employed to mitigate the negative effects of garbage collection [14, 18, 19] and read disturb [21] in RAID-enabled SSDs or SSD-based RAID storage. Specifically, Yan et al. [14] designed *Tiny-tail flash* that aims to service GC-blocked I/Os by exploiting parity-based redundancy to proactively generate required contents. Then, it can yield (nearly) GC-free I/O latency. Cui et al. [21] introduced a disturb-aware



(a) Read Latency (ms) of *MSRC wdev_0* (b) Read Latency (ms) of *lun1-0* (c) Read Latency (ms) of *lun3-0*

Fig. 2. CDF of I/O latency on the most congested channel and the remainder channels of the same stripes in conventional RAID-enabled SSDs. The number of blocked requests on channels is shown in the legend.

read redirection scheme. It also makes use of parity-based redundancy to regenerate data contents, to avoid reading the target data chunk that may cause read disturb errors.

Existing work commonly makes use of empirical rules to decide if parity-based redundancy should be used for regenerating the expected data chunks. For example, *Tiny-tail Flash* checks whether the relevant RAID components (i.e. SSD channels) are currently enduring a GC operation or not, to decide whether triggering an optimization read operation or not [14]. We argue that it is better to have a common model to direct triggering degraded modes for satisfying read or update requests for eventually yielding I/O workload balance among all SSD channels, by considering not only whether channels are busy on GC operations or not, but also the levels of busyness of involved SSD channels caused by serving normal I/O requests.

2.2 Motivations

To assess how the factor of channel busyness affects the response latency to I/O requests, we measured the I/O response time for different channels of the same stripes. We recorded that average number of blocked requests and the cumulative distribution function (CDF) of the I/O latency on different SSD channels after replaying the selected three block traces on our experimental platform. See Section 4.1 for the details on the benchmarks and the experimental platform. Specially, we used **additional**-01-2016021615-LUN0 (labeled as *lun1-0*), and **additional**-03-2016021715-LUN2 (*lun3-0*), from the LUN trace collection [30].

Figure 2 shows the relevant results. In the figure, the horizontal axis represents the different disk traces of real world applications, and the vertical axis indicates the CDF value of I/O latency. Clearly, varied SSD channels in the same stripe have different long-tail latency on the channels. Besides, we measured the average numbers of blocked requests in I/O queues of channels, and disclosed that a large tail latency generally corresponds to a large number of blocked I/O requests on the channel. More importantly, we observed that except for the most congested channel that has the worst long-tail latency, other channels in the same stripe have less long-tail latency with a similar tendency.

Such observations motivate us to propose a novel I/O scheduling mechanism for achieving an I/O workload balance across all channels of RAID-enabled SSDs, to better ensure I/O responsiveness of user applications.

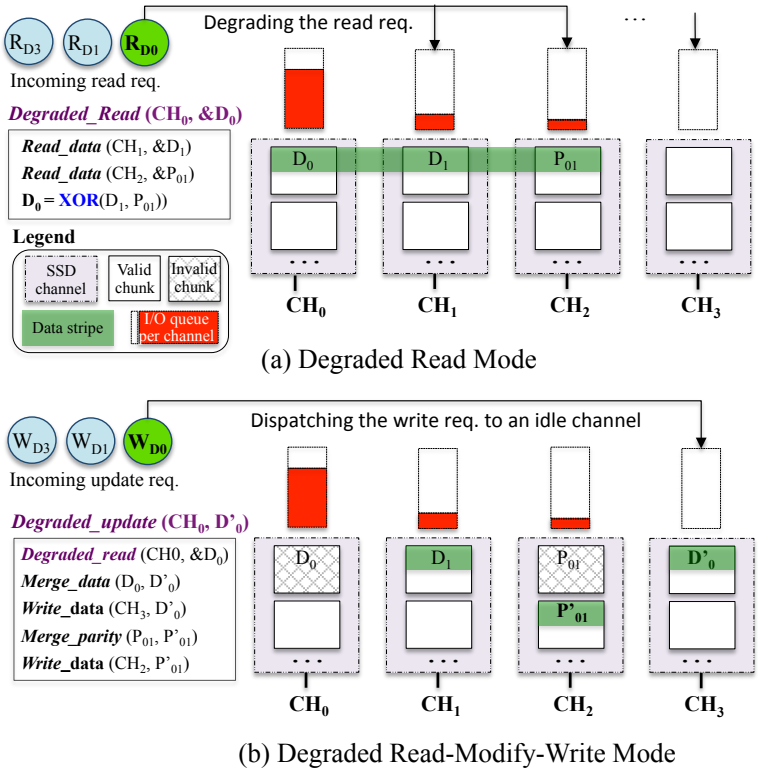


Fig. 3. High Level Overview of Degraded Read Mode (a) and Degraded Read-Modify-Write Mode (b) in RAID-enabled SSDs.

3 DEGRADED MODE-BASED SCHEDULING

3.1 Architectural Overview

The basic principle of degraded mode-based scheduling is to avoid dispatching read requests or update requests to the most congested but targeted SSD channels. Then, such requests will be serviced with the degraded mode by accessing other SSD channels in the same data stripe. Figure 3 illustrates a high-level overview of both degraded read and read-modify-write (i.e. update) modes, for possibly satisfying certain read/update requests.

Figure 3(a) describes the degraded read mode. As seen, the read request of R_{D0} should be dispatched into the queue of CH_0 which is very busy, according to the read/write rules in RAID-enabled SSDs. But in our context, the queuing overhead assessment model (described in Section 3.2) detects that CH_0 would take too much time to satisfy this request. Thus, we combine D_1 on CH_1 and P_{01} on CH_2 with a XOR operation for regenerating the contents of D_0, to eventually avoid worsening of the congestion state of I/O queue on CH_0. The semantics of the degraded read function (defined as `Degraded_read()`) consist of two read operations and one XOR operation, which are also illustrated in Figure 3(a).

Figure 3(b) shows how the degraded update mode satisfies an update request of W_{D0} (i.e. a read-modify-write request), which originally targets at CH_0, the most congested channel. Then, it first obtains the obsolete data chunk of D_0 by employing a degraded read operation. Next, it generates the new data chunk by merging the old data chunk with the new coming data. After that, it tries to adaptively organize the stripe by flushing the new data chunk onto a not busy

Table 1. Symbols and Explanations in the Queuing Overhead Assessment Model

Symbol	Explanation	Symbol	Explanation
CH_i	the ith channel	f_i	arriving req.
R_i	read on CH_i		freq. on CH_i
W_i	write on CH_i	t_{rd}	read latency
n_i	enqueue req. on CH_i	t_{wr}	write latency
n_{ir}	enqueue reads on CH_i	t_{gc}	GC latency
n_{iw}	enqueue writes on CH_i	t_{xor}	XOR latency
t_{wr_p}	write latency on parity		

channel (i.e. CH_3), for the purpose of balancing data access workloads across all channels in SSDs. At last, the parity chunk are re-computed and then updated in the same channel (CH_2), as this channel is not the busiest one at that time. The detailed semantics of the degraded read-modify-update function (defined as `Degraded_update()`) have been shown in Figure 3(b).

3.2 Queuing Overhead Assessment Model

To precisely guide whether dispatching I/O requests to be serviced with the degraded mode or not, we build a queuing overhead assessment model. Table 1 summarizes the notations and the corresponding definitions used in our model.

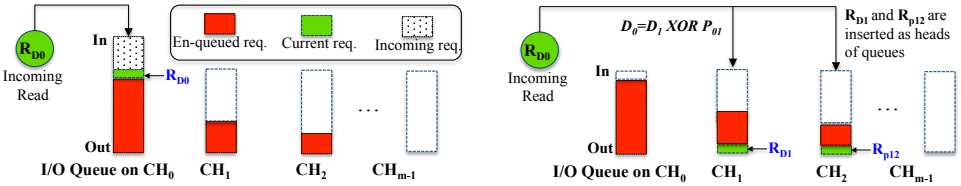
Note that our approach adopts stripe structures that do not fully span all channels. In fact, early SSDs usually have a limited number of channels, such as 4, so that it is common to make the stripe size matching the number of channels. Nowadays high density SSDs, however, usually have more channels [22, 23]. For example, the *SM8266* controller enables 16 NAND flash channels, and is designed to support the latest 3D TLC and QLC NAND flash technologies [24]. Then, building a stripe structure by using only a part of channels (or chips in chip-level RAID implementations) has been used in practice, to avoid write amplification due to the over-large stripe structure [25]. In our context, supposing the RAID-enabled SSD has m channels of CH_0, CH_1, ... and CH_{m-1}, and the first k channels are organized as the stripe structure. That is, each data stripe includes $k-1$ data chunks and 1 parity chunk ($2 < k < m$).

To simplify descriptions on modeling, we assume the read request of R_{D0} arrives to fetch the data chunk managed by CH_0, and D_0, D_1, and P_{01} form a data stripe (size $k = 3$). There are two ways to service this request. One selection is to normally enlist R_{D0} to the waiting queue of CH_0, and another is to read the remainder chunks in the same stripe (i.e. CH_1 and Ch_2) to regenerate the expected data with certain XOR computations.

From the perspective of satisfying the current request of R_{D0}, the relevant waiting overhead consists of two parts of T_{wait} and T_{delay} if we use the normal read mode (see Figure 4(a)). Specifically, T_{wait_a} is the waiting time of R_{D0} that is caused by completing the en-queued requests on the same channel, and defined in Equation 1.

$$T_{wait\_a} = n_{0r} \cdot t_{rd} + n_{0w} \cdot t_{wr} + c \cdot t_{gc} \tag{1}$$

where c could be 0, 1, 2, ..., and indicates the number of GCs during the period of completing I/O requests on CH_0. This parameter can be estimated by the current available space and the amount of write data induced by the en-queued write requests.



(a) Normal Model for Read (b) Degraded Model for Read

Fig. 4. Servicing a Read request with the Normal mode (a) and the Degraded mode (b) impacts I/O queuing on channels.

Meanwhile, there will be certain future incoming requests enter into the I/O queue of CH_0 before servicing R_{D0}. We label the number of incoming requests as N_0, that is related to the arriving request frequency and the time interval of T_{wait_a}:

$$N_0 = f_0 \cdot T_{wait\_a} \tag{2}$$

Then, each of these incoming requests must be delayed by one more read latency caused by servicing R_{D0} (i.e. t_{rd}), and we label this part of time cost as T_{delay_a}. Then, we define the overall waiting cost of dispatching the request of R_{D0} with the normal mode onto the target channel in Equation 3.

$$\begin{aligned} T_A &= T_{wait\_a} + T_{dealy\_a} \\ &= T_{wait\_a} + t_{rd} \cdot N_0 \\ &= T_{wait\_a} + t_{rd} \cdot (f_0 \cdot T_{wait\_a}) \\ &= (n_{0r} \cdot t_{rd} + n_{0w} \cdot t_{wr} + c \cdot t_{gc}) \cdot (1 + t_{rd} \cdot f_0) \end{aligned} \tag{3}$$

On the other side, if R_{D0} is completed with the degraded read mode (see Figure 4(b)), two associated read requests will be additionally inserted into the heads of queues of CH_1 and CH_2 for fetching the remainder chunks of the same stripe. After that, an XOR operation will be used to regenerate the expected data for R_{D0}. Because two newly inserted read requests will be serviced immediately, we argue that they will not have the waiting time cost. Then, these inserted read requests only bring about impacts on the currently en-queued requests on the involved channels. Considering we intend to yield a minimum of overall queuing time of all channels, the overhead of responding the data with the degraded read mode is defined Equation 4.

$$\begin{aligned} T_B &= t_{xor} + T_{wait\_b} + T_{delay\_b} \\ &= t_{xor} + T_{delay\_b} \\ &= t_{xor} + \sum_{i=1}^{k-1} T_{delay\_b\_i} \\ &= t_{xor} + \sum_{i=1}^{k-1} t_{rd} \cdot n_i \\ &= t_{xor} + t_{rd} \cdot \sum_{i=1}^{k-1} n_i \end{aligned} \tag{4}$$

where $T_{delay\_b\_i}$ is the total increased waiting overhead on the ith channel, caused by inserting a read request (i.e. with the degraded read mode) into the channel.

As seen, the model needs to obtain the total waiting time on all relevant channels caused by the degraded read operation for eventually yielding an I/O workload balance, though SSD channels can work in a parallel manner. Then, it can compare T_A with T_B when scheduling the read request of R_0, and then select the degraded read mode if T_B is the smaller one. Otherwise, the read request of R_0 keeps waiting on the target channel of CH_0.

With respect to servicing the update request of W_0 that modifies the original data chunk managed by CH_0, we first estimate the overhead of using the degraded mode and the normal mode. Similarly to the case of read requests, Equations 5 defines the waiting overhead of completing the update request on CH_0.

$$\begin{aligned} T'_A &= T_{wait\_a} + T'_{dealy\_a} + t_{wr\_p} \\ &= (n_{0r} \cdot t_{rd} + n_{0w} \cdot t_{wr} + c \cdot t_{gc}) \cdot (1 + t_{wr} \cdot f_0) + t_{wr\_p} \end{aligned} \tag{5}$$

where t_{wr_p} means the time of writing the parity channel, which is equal to t_{wr}.

Equation 6 defines the waiting overhead of satisfying the request with the degraded update mode.

$$T'_B = t_{xor} + t_{rd} \cdot \sum_{i=1}^{k-1} n_i + t_{wr} \cdot n_x + t_{wr\_p} \tag{6}$$

where n_x is the number of waiting requests on the xth channel that is the most idle but not associated with the W_0's stripe.

If T'_B is smaller than T'_A, we select the degraded read-modify-write mode to complete the write request. That is, the updated chunk is placed on a slack channel that was not a part of the stripe, to achieve not only a quick write completion but also a balanced I/O workload distribution. If not, the request of R_0 keeps waiting on the target channel of CH_0.

3.3 Implementation Issues

ALGORITHM 1: Cost assessment model-based scheduling

Input: args of n_{jr}, n_{jw}, f_j on stripe-involved k channels (e.g. CH_0 to CH_{k-1} for algorithm illustration);
Output: the degraded mode or not for Req_i on CH_i;

```
/*Initializing the overhead of both scheduling routines*/
T_A = T_B = 0;
if Req_i is Read then
    T_A = (n_ir · t_read + n_iw · t_write + t_gc) · (1 + t_read × f_i);
    T_B = t_xor + t_read · Σ_{j=0,j≠i}^{k-1} (n_jr + n_jw);
    if T_A > T_B then
        degraded_read(Req_i); //degraded read
    else
        common_read(Req_i); //keep waiting on CH_i
else
    T_A = (n_ir · t_read + n_iw · t_write + t_gc) · (1 + t_write × f_i) + t_wr_p;
    /*No. x channel has the least number of requests and not involved in the target stripe.*/
    T_B = t_xor + t_read · Σ_{j=0,j≠i}^{k-1} (n_jr + n_jw) + t_write · n_x + t_wr_p;
    if T_A > T_B then
        degraded_update(Req_i); //degraded update
    else
        common_update(Req_i); //keep waiting on CH_i
```

Table 2. Experimental settings of *SSDsim*

Parameters	Values	Parameters	Values
Channel Size	8/16	*Read latency*	0.045ms
Chip Size	4	*Write latency*	0.7ms
Plane Size	4	*Erase latency*	3.5ms
Block per plane	256	*XOR latency*	0.019ms
Page per block	128	*GC Threshold*	10%
Page Size	8KB	*RAID Level*	5
FTL Scheme	Page	*ECC*	LDPC (7 levels)
Wear-leveling	Static	*Stripe Struct.*	3 Data + 1 Parity

Algorithm 1 shows the implementation details on the proposed I/O dispatching scheme with the queuing overhead assessment model. As read, Lines 3-9 cope with a read request. Specifically, it computes the overhead of two scheduling routines (i.e. degraded read and common read) by referring to the model, and then selects the one having less overhead. Similarly, Lines 11-17 present the details of dealing with an update request.

4 EXPERIMENTS AND EVALUATION

This section first describes the experimental setup. Then, evaluation results and relevant discussions are presented, to show the feasibility and applicability of degraded mode-benefited I/O scheduling. At last, we make a brief summary about the findings obtained from the evaluation experiments.

4.1 Experimental Setup

The SSD controller has limited computation power and memory capacity [26], we then carried out tests on a local ARM-based machine. It has an ARM Cortex A7 Dual-Core with 800MHz, 128MB of memory and 32-bit Linux (*kernel ver 3.1*). We have performed trace-driven simulation by replaying block I/O traces of real-world application with *SSDsim (ver2.1)*. Because *SSDsim* has a diverse set of configurations, it has been widely employed in many studies for measuring SSD performance through running simulation tests [27]. Table 2 demonstrates our settings of *SSDsim* in experiments, which have been also used in prior studies [28, 29]. In the evaluation tests, we did emulate a DRAM-less SSD device (64GB and 128GB) that does not hold a specific cache for buffering the write data.

Specially, we recommend computing the frequency of I/O requests (i.e. f_i in the model) by referring to recent 128 requests on the channel by default, after certain preliminary tests on our platform. the GC time (i.e. the parameter of t_{gc} in the model) consists of the time required for completing page moves and the time of erase. Although the number of page moves in each GC is not many and relatively stable, we set the value of this parameter being changeable, by following Equation 7.

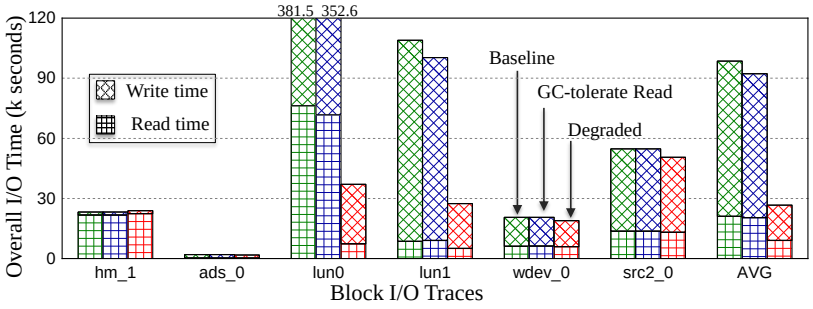
$$t_{gc} = (Read\ latency + Write\ latency) \times Avg\ Number\ of\ page\ moves + Erase\ latency \tag{7}$$

where $Avg\ Number\ of\ page\ moves$ means the average number of page moves in recent 8 GC operations on the channel.

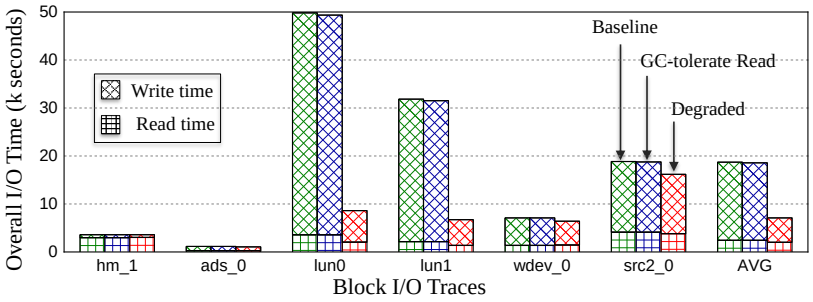
Moreover, we argue that the proposed degraded update mode works for the contexts in which the number of channels is larger than the size of stripe (discussed in Section 3). To check the effectiveness of our proposal in modern high density SSDs, we conduct evaluation tests with SSD configurations of 8 and 16 channels respectively.

Table 3. Specifications on I/O Traces (ordered by *Wr Ratio*)

Traces	Req #	Wr Ratio	Avg Req SZ	Max/Avg INT
hm_1	609311	4.6%	15.1KB	144.0/9.8ms
ads_0	1532120	9.50%	29.1KB	224.1/5.6ms
lun0	3316282	47.6%	16.5KB	158.1/1.1 ms
lun1	2667824	52.1%	16.5KB	164.0/1.3 ms
wdev_0	1143261	79.9%	6.6KB	525.6/5.3ms
src2_0	1557814	88.7%	6.3KB	7822.1/4.5ms



(a) 8 SSD channels



(b) 16 SSD channels

Fig. 5. Overall I/O time of running the benchmarks with configurations of 8 and 16 SSD channels.

In order to diversify the benchmarks and make long runs of simulation, we have selected a number of block I/O traces of real applications from different research groups, and repeated them twice in our tests. Table 3 reports the specifications of the selected traces. Among them, two merged block I/O traces are recently collected from a part of an enterprise VDI (Virtual Desktop Infrastructure) [30, 31]. Specifically, we have merged four consecutive traces starting from ***additional-01-2016021616-LUN0*** to form *lun0*, and four consecutive traces starting from ***additional-01-2016021617-LUN0*** to form *lun1* in our tests. Three traces (*hm_1*, *src2_0* and *wdev_0*) are from Microsoft Research and afforded by multiple enterprise servers for different real-world applications [32]. The trace *ads_0* comes from [33] the set of Microsoft Production Server Traces.

Besides our proposal, another two schemes were also used as comparison counterparts in evaluation tests:

Table 4. Number of degraded operations (XOR operations)

Block traces	***GC-tolerate Read***		***The Degraded scheme***	
	read #	*update #*	*read #*	*update #*
*hm_1 (**8 CH.**)*	0	-	3300	176
ads_0	22	-	36610	912
lun0	10696	-	64457	59690
lun1	8880	-	42790	68737
wdev_0	981	-	50682	15947
src2_0	1343	-	87205	32191
*hm_1 (**16 CH.**)*	0	-	4757	191
ads_0	0	-	30221	3876
lun0	9212	-	22916	4462
lun1	6285	-	27369	61666
wdev_0	868	-	60729	16707
src2_0	1633	-	86541	23120

Note: ***GC-tolerate Read*** does not support the degraded update mode.

- ***Baseline***, which implies that RAID-enabled SSDs only support conventional I/O scheduling. In other words, both read and update requests have to be dispatched onto the target channels, regardless their congestion status.
- ***GC-tolerate Read***, which can fulfill read requests targeting at the GC-blocked channel, by exploiting parity-based redundancy to proactively generate the required contents [14]. We argue that it is the most related work to our proposal, as it can speedup responding to the read requests on GC-blocked channels though it does not intend to yield a balanced I/O workload distribution among all SSD channels.
- ***Degraded***, which is the proposed I/O scheduling approach. It has a queuing overhead assessment model to precisely satisfy a part of read and update requests on hot channels by using degraded modes in RAID-enabled SSDs, for the purposes of reducing the long-tail latency and achieving I/O workload balance.

4.2 Results and Discussions

To evaluate the validity of the proposed scheduling approach for RAID-enabled SSDs, we take advantage of the metrics of *overall I/O time* and *long-tail latency* to reflect the I/O responsiveness while running the benchmarks. After that, we analyze GC statistics and workload balance over SSD channels, as well as the space and computation overhead.

4.2.1 Overall I/O Time. We have measured the time required for replaying the block I/O traces by using the chosen scheduling methods. Figures 5(a) and 5(b) present the results of overall I/O time that consists of the read time and the write time, while the SSD device has 8 and 16 channels respectively. In contrast to *Baseline*, both *GC-tolerate Read* and *Degraded* can noticeably reduce the overall I/O latency of all traces, as *Baseline* does not support I/O redirection optimization, even though a part of channels are heavily congested with I/Os or GC operations.

More exactly, *Degraded* noticeably improves the overall I/O time by 38.7% on average in all traces, compared with the most related work of *GC-tolerate Read*. This is because *GC-tolerate Read* only fulfills read requests with the degraded mode if the target channel is busy on GC. But, the proposed queuing overhead assessment model takes both

GC operations and the workload of each channel into account, while servicing both read and update requests. Thus, our *Degraded* strategy triggers more degraded operations, which improves I/O performance. As reported in Table 4, it triggers 20.1 times more degraded operations on average, compared with *GC-tolerate Read* after running the benchmarks. Although the degraded I/O scheduling method redirects a write request from the original channel to another idle channel, it does not affect the endurance span of SSDs since no more write operations are introduced.

Besides, we see the overall I/O time keeps decreasing when the number of SSD channels changes from 8 to 16, this is due to the I/O contention is reduced to some extent if the SSD device has more channels for further supporting parallel I/O accesses. Another important clue illustrated in the figure is that *Degraded* can scale well in the configurations with more SSD channels, but *GC-tolerate Read* does not noticeably work better than *Baseline* in the case of 16 channels. This is because *GC-tolerate Read* can only speedup the processing on the blocked I/O requests caused by GC operations through accessing other relevant SSD channels. But, the 16-channel SSD has more capacity and endures less GC operations comparing with the 8-channel SSD (refer to Section 4.2.3 for details), so that the benefits of *GC-tolerate Read* are confined.

In brief, the I/O improvements brought about by *GC-tolerate Read* are tightly related to the number of GC operations. But our *Degraded* proposal can achieve the same level of I/O enhancements in a wide range of SSD capacity configurations, through evening I/O workloads over all SSD channels.

4.2.2 Long-tail Latency. As discussed in our motivations, imbalanced I/O workloads on RAID-enabled SSD channels must postpone some I/O requests and then worsen the tail latency problem. Considering write requests generally do not expect as much response latency as read requests [35], we only collect the Cumulative Distribution Function (CDF) of read latency after running all benchmarks. Figures 6 and 7 illustrate the CDF results of read latency when using the selected scheduling methods, with configurations of 8 and 16 SSD channels respectively.

Clearly, the lines of *Baseline* are almost the lowest ones since it does not adopt any optimization strategies to decrease the long-tail latency. Our proposed *Degraded* method exhibits better long-tail latency than that using other comparison schemes. More exactly, our proposal can cut down the long-tail latency by up to 98.9% (i.e. 23.1% on average) at the 99.99th percentile compared with *GC-tolerate Read*. This fact proves that avoiding scheduling read or update requests to the heavily congested SSD channels can efficiently cut down the long-tail latency.

It is worth to mention that our proposed *Degraded* scheme slightly perform worse than *Baseline* and *GC-tolerate Read* when replaying *hm_1*. This is because *hm_1* is a read-intensive application and has an even I/O workloads across all channels, we cannot benefit from the degraded read/write mode offered by the *Degraded* approach. In fact, we reported that *Degraded* causes a small increase in the overall I/O time after running the *hm_1* trace, refer to Section 4.2.1.

4.2.3 GC Statistics and Analysis. Both proposed *Degraded* method and *GC-tolerate Read* take the GC processes into account when issuing degraded operations. Therefore, we record the number of GC operations after replaying the selected block traces with varied scheduling methods, and the results are demonstrated in Figure 8.

As seen in the figure, the proposed *Degraded* scheme can slightly cut down the number of GC operations by 349.1 on average comparing with *Baseline* and *GC-tolerate Read*, after running a major part of traces. On the one hand, both *Baseline* and *GC-tolerate Read* place the updated data chunks in the same channels, which must result in more GC operations on the channels that have hot write chunks. According to our measurements of coefficient of variation (*cv*) for GC counts of all SSD channels, we see the value of *cv* varies from 0.3 to 2.3 after replaying all the traces with *Baseline* and *GC-tolerate Read*. That is to say, the GC count on varied channels differs from each other greatly, so that we obtain a relative large GC count in total.

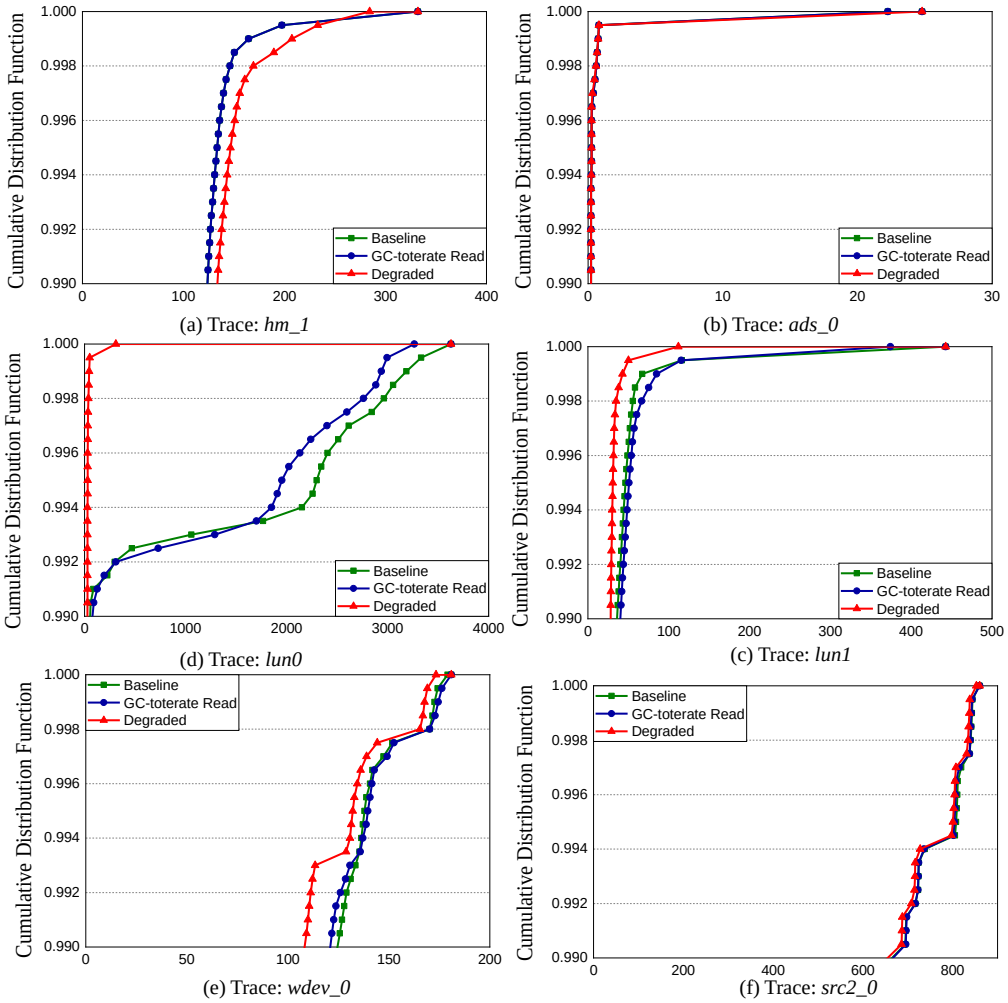


Fig. 6. Long-tail latency of read requests with 8 channels (unit: ms).

On the other hand, *Degraded* basically places the update data onto the non-busy channels and causes not much differences on the free space of channels. More exactly, the value of *cv* of GC count ranges from `0.01` to `0.3` after replaying all the traces when using *Degraded*. As a result, *Degraded* brings about a relative small GC count after replaying the selected traces.

In addition, we understand that *GC-tolerate Read* brings about less than `21.9%` of GCs in the configuration of `16` channels, comparing with the configuration of `8` channels. The GC operations are triggered when the free space of SSDs is smaller than a predefined threshold and `16` SSD channels indicate more storage capacity, so that the number of GC operations may be consequently cut down with the same amount of write data after running the traces. As a result, *GC-tolerate Read* achieves better I/O performance in a small size of SSD capacity, and we can conclude *GC-tolerate Read* is GC-dependent and the number of GC operations greatly affect I/O improvements.

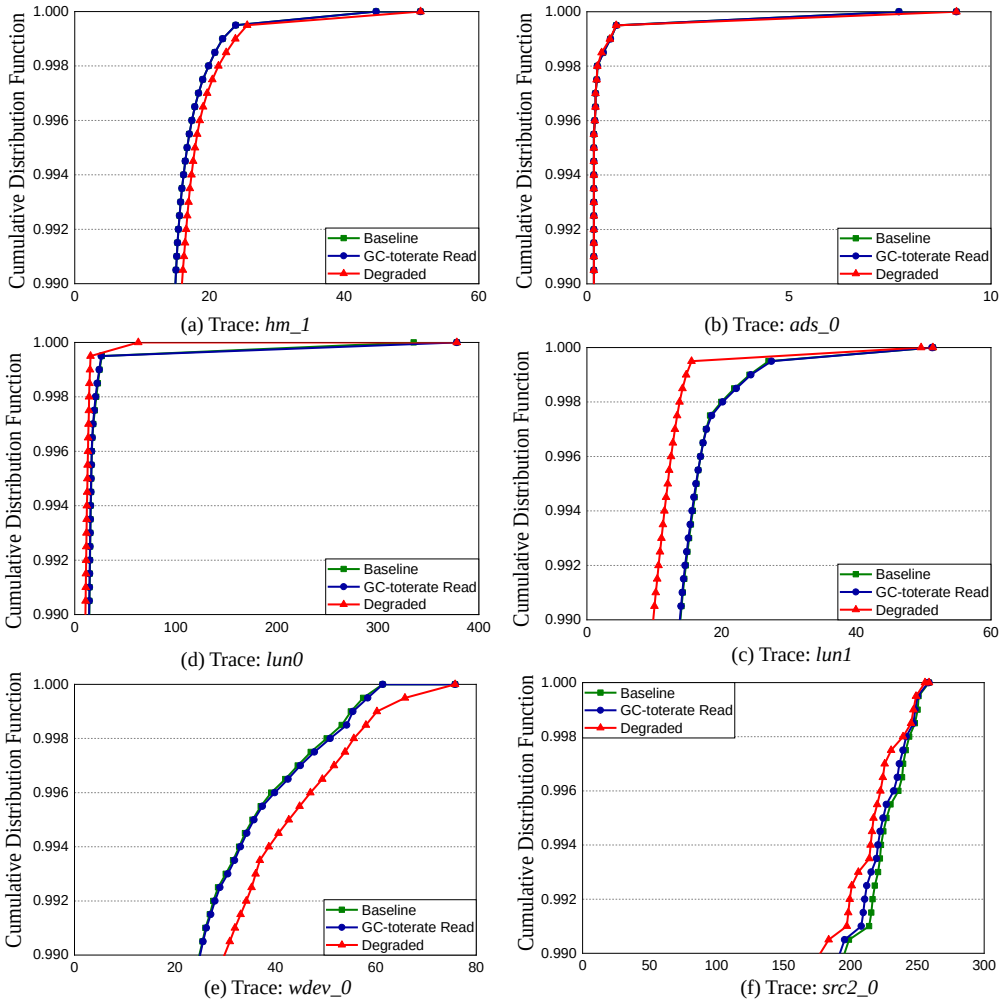


Fig. 7. Long-tail latency of read requests with 16 channels (unit: ms).

4.2.4 Model Verification. To better verify how our proposed model yields evenly distributed I/O workloads among all channels of RAID-enabled SSDs, we carry out workload and congestion analysis by recording the average completed requests and the average blocked requests per channel. Specifically, the average number of blocked requests is defined as the number of blocked requests (i.e. we count the blocked requests per request processing and then make a sum), divided by the number of completed requests on a specific channel.

Figure 9 shows the average completed requests per channel after replaying all selected I/O traces with the configurations of 8 and 16 channels. From both sub-figures, we see that the absolute number of completed requests per channel keeps decreasing when the number of SSD channels becomes larger, but the comparison trend of three scheduling approaches is similar. More specifically, compared with *Baseline*, *GC-tolerate Read* and *Degraded* cope with more requests by 0.1% and 0.5% on average. This is because the degraded mode may issue multiple requests onto other related

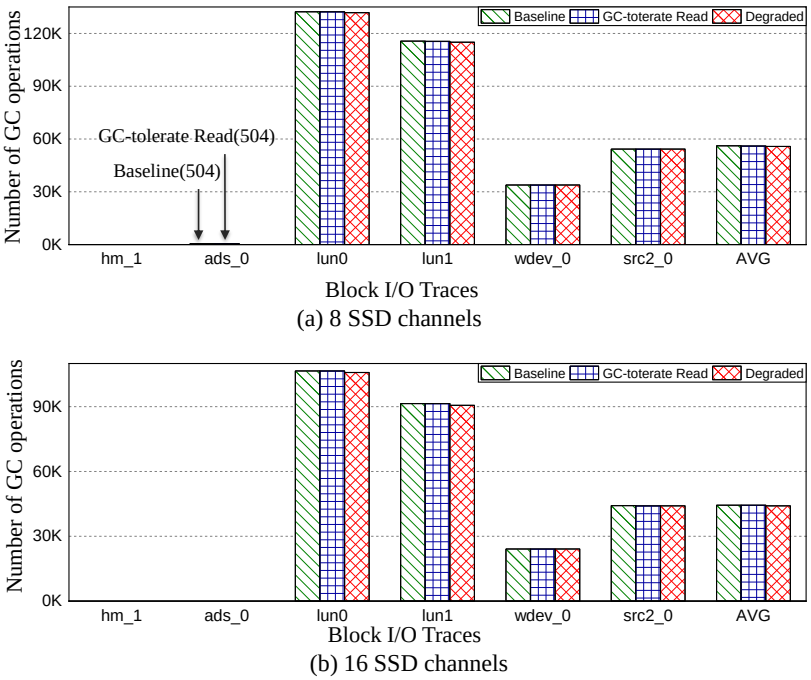


Fig. 8. The results of GC statistics.

channels for servicing one read/update request. Another important information is that, our proposal of *Degraded* yields the best workload balance among all channels, as it has the smallest standard deviation. In other words, the proposed queuing overhead assessment model in *Degraded* can noticeably contribute to the balanced workloads (read/write/GC) among all SSD channels.

Figure 10 presents the results of average blocked requests per channel. *Degraded* achieves the least number of blocked requests per channel in all cases. This is because our method uses the degraded read/update mode to mitigate the congestion status of hot channels. Consequently, the I/O requests are evenly distributed across all channels, as shown by our proposal that has the least standard deviation of blocked requests per channels.

Our model empirically uses Equation 7 to estimate the critical parameter of the GC time for guiding I/O scheduling. Figure 11 presents the comparison of the estimated GC time and the real GC time after replaying the traces. Note that *Degraded* does not introduce any GC operations when running the read-intensive traces of *hm_1* and *ads_0*. As illustrated, the estimated GC time does not noticeably differ from the time needed in the real GC process. This is because a GC operation consists of a fixed *Erase* and a small indeterminate number of *Page Move*s (i.e. on average, 5.5 page moves/GC in our tests), and the *Erase* operation takes a major part of time overhead in a GC process.

4.2.5 Overhead Analysis. The main space overhead is caused by holding the mapping table, and recording the parameters used by the queuing cost assessment model. On the one hand, the degraded mode needs recording the locations of channels that hold chunks associating with the given stripe. Thus, it requires more memory space for keeping the 2-level mapping table, by additionally recording the channel location for each (data/parity) chunk. More exactly, our approach

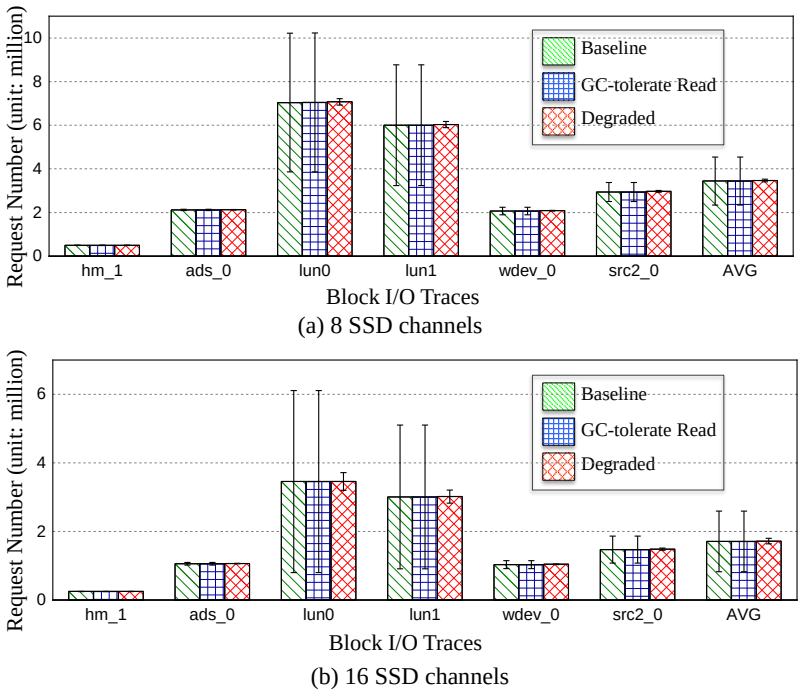


Fig. 9. The number of completed requests. Note that the error bar indicates the standard deviation among all SSD channels.

needs up to 132MB = (128×1024×32) chunks × ((8B) stripe No. +(0.5B) channel No. + (8B) offset). Comparing with *Baseline* and *GC-tolerate Read*, which need 128MB space in the configuration of 16 channel, our method requires more space overhead by 3.1%.

On the other hand, the queuing cost assessment model records the timestamps of 128 recent requests and the blocked read and write counts, for each channel. Thus, it needs up to 16.2KB = 16 channels × (128 timestamps × 8B + 2 counts × 4B). In brief, our proposal demands a reasonable amount of memory space in SSDs.

As previously reported in Table 4, both degraded modes considerably amplify the number of reads and writes to RAID components, and these amplified reads and writes all need to go through the error correction codes (ECC) component of SSD for fighting against read disturb [34]. Then, the computation overhead of our approach consists of the time needed for performing extra ECCs on amplified I/O operations caused by degraded modes, the time expected for completing XOR operations, and the time required for estimating the queuing overhead for an incoming request.

Though the related work of *GC-tolerate Read* does have certain amplified read requests caused by degraded read operations, these requests do not result in additional time caused by higher-level ECC corrections. On the other side, our *Degraded* proposal may cause extra time for higher-level ECC corrections after running the benchmarks of *lun1-2* and *lun1-1*. Figures 12(a) and 12(b) present the results of computation overhead of our proposed method in the configurations of 8 and 16 channels respectively.

As shown in Figure 12(a) with the case of 8 channels, *Degraded* causes time overhead between 0.7 and 8.8 seconds after replaying the selected traces. This corresponds an average of 4.9μs per I/O request, or less than 0.4% of the overall I/O time. In the case of 16 channels reported in Figure 12(b), we see its computation overhead is greatly less than

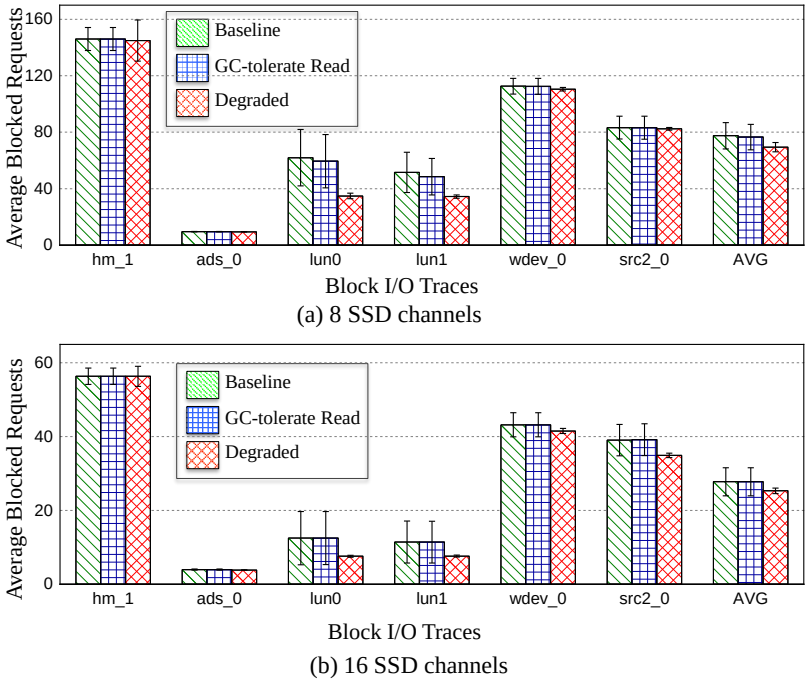


Fig. 10. The number of blocked requests per SSD channel. Note that the error bar indicates the standard deviation among all SSD channels.

the configuration of 8 channels. This is because the amplified reads and writes issued by degrade modes are distributed onto all 16 channels, which do not introduce higher-level ECC corrections in SSD blocks for reliability enhancement, compared with *Baseline*. Then, we conclude that *Degraded* has less time overhead in the configurations of more SSD channels, since the amplified read/write requests can be dispatched onto more SSD channels for relieving the side-effect of read disturb.

In summary, we consider that the time overhead caused by degraded mode-based I/O scheduling is acceptable, even though our tests are conducted on an ARM-based platform. Note that the computation overhead does bring about impacts on I/O response time by postponing dispatch on incoming I/O requests. The results of overall I/O time, the SCB score, and the long-tail latency reported in Sections 4.2.1 to 4.2.2 have considered the impact of computation overhead.

4.3 Sensitive Analysis on Stripe Size

This case study checks the sensitivity of our approach on the stripe size, and we span the stripe structure with 3+1, 5+1, and 7+1 channels. Figure 13 shows the results of I/O latency after running the benchmarks under different size of stripe structure, in the configuration of 16 SSD channels.

The proposed *Demand* scheme can save I/O latency by more than 7.4%, in contrast to other comparison counterparts. This verifies that our approach can scale well on the different size of stripe structure, as our proposal supports both degraded read and write scheduling. In general, a large size of stripe structure can contribute to the reduction of parity updates while the workloads have many big write requests (e.g. *lun0* and *lun1*), but it increases the number of

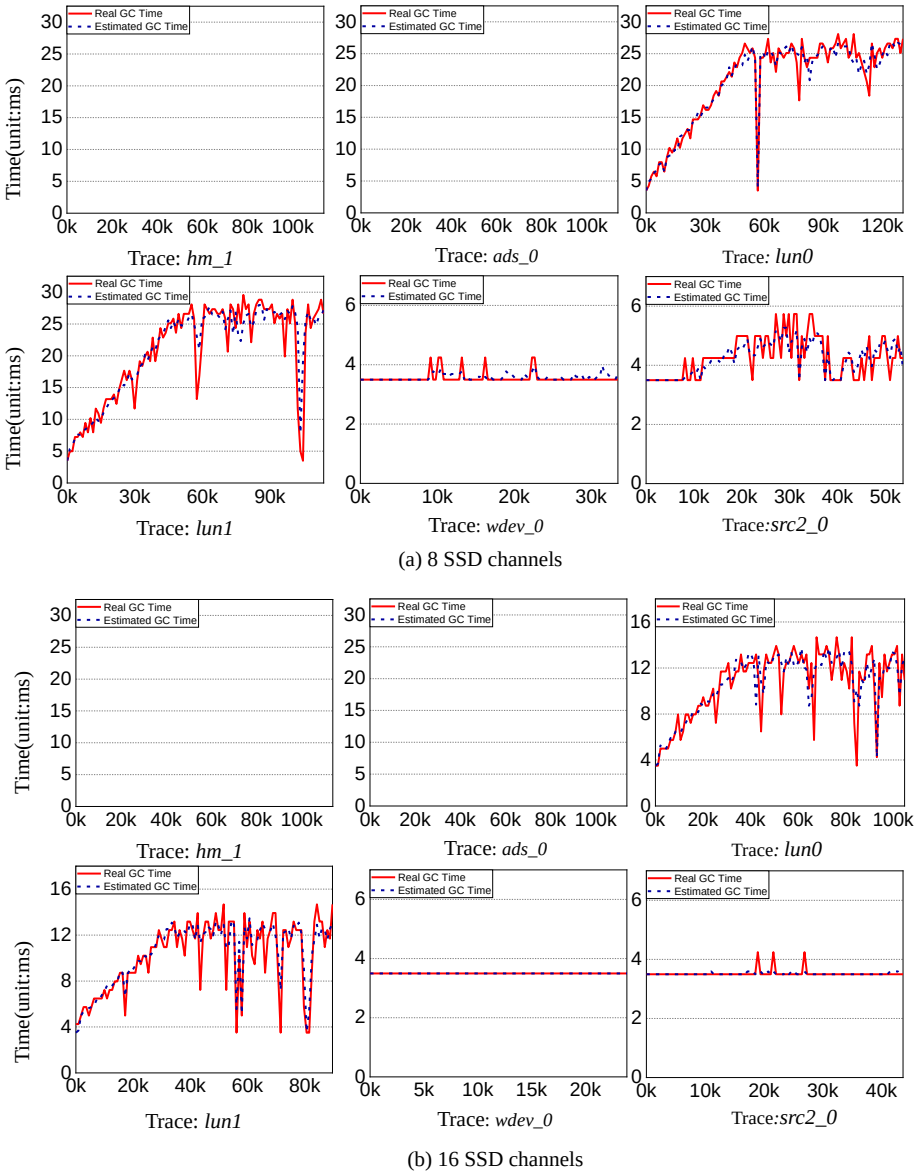
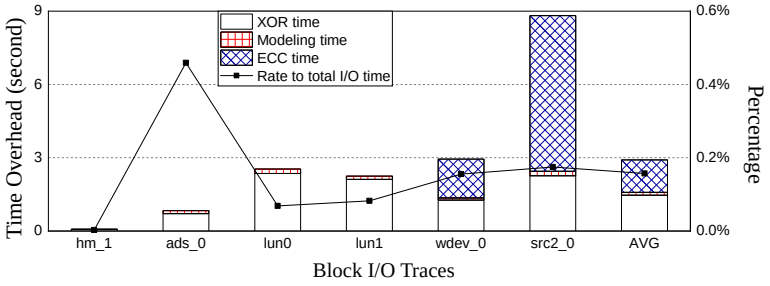


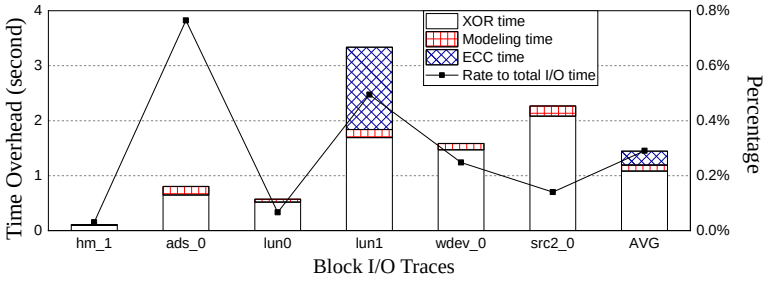
Fig. 11. Verification of the accuracy of estimated GC time.

parity updates while the majority of write requests are small ones (e.g. *wdev_0* and *src2_0*). Furthermore, a large stripe structure implies more associate read requests are supposed to be generated to fulfill a read request with the degraded read mode.

Another interesting result is that the related work *GC-tolerate Read* does not noticeably outperform *Baseline*, and it even performs worse when running *lun0*. We think that this is because the 16-channel configuration of SSDs has



(a) 8 SSD channels



(b) 16 SSD channels

Fig. 12. Time overhead of *Degraded* after running benchmarks.

128GB capacity, which does not require many GC operations to reclaim space and thus limits the room for performance improvement of *GC-tolerate Read*. In addition, *GC-tolerate Read* needs to issue much more associate read requests for serving a read request with the degraded mode, when the stripe structure becomes larger.

4.4 Summary

With respect to comparing conventional I/O scheduling approaches in RAID-enabled SSDs and the proposed scheduling scheme, we emphasize the following two key observations. ***First***, with the proposed queuing overhead assessment model, we can ideally migrate I/O requests from busy channels to idle channels, for balancing I/O workloads across all RAID components and for a better I/O performance. ***Second***, with the supports of degraded read and read-modify-write modes, the migrated I/O requests can be fulfilled by accessing on slack channels, so that the long-tail latency of I/O requests in RAID-enabled SSDs can be significantly cut down. More importantly, *Degraded* is not GC-triggered and can also yield noticeable I/O improvements in the cases of not many GC operations.

5 CONCLUSION

This paper proposes an I/O scheduling approach for RAID-enabled SSDs to yield a balanced I/O workload distribution and then to ensure their I/O responsiveness, via using degraded modes to selectively fulfill read/update requests. To this end, it first builds a mathematical queuing cost assessment model by referring to the data redundancy and the blocked I/O traffics on the different RAID components. Then, it can determine whether the I/O requests are supposed to be serviced with the degraded modes or not.

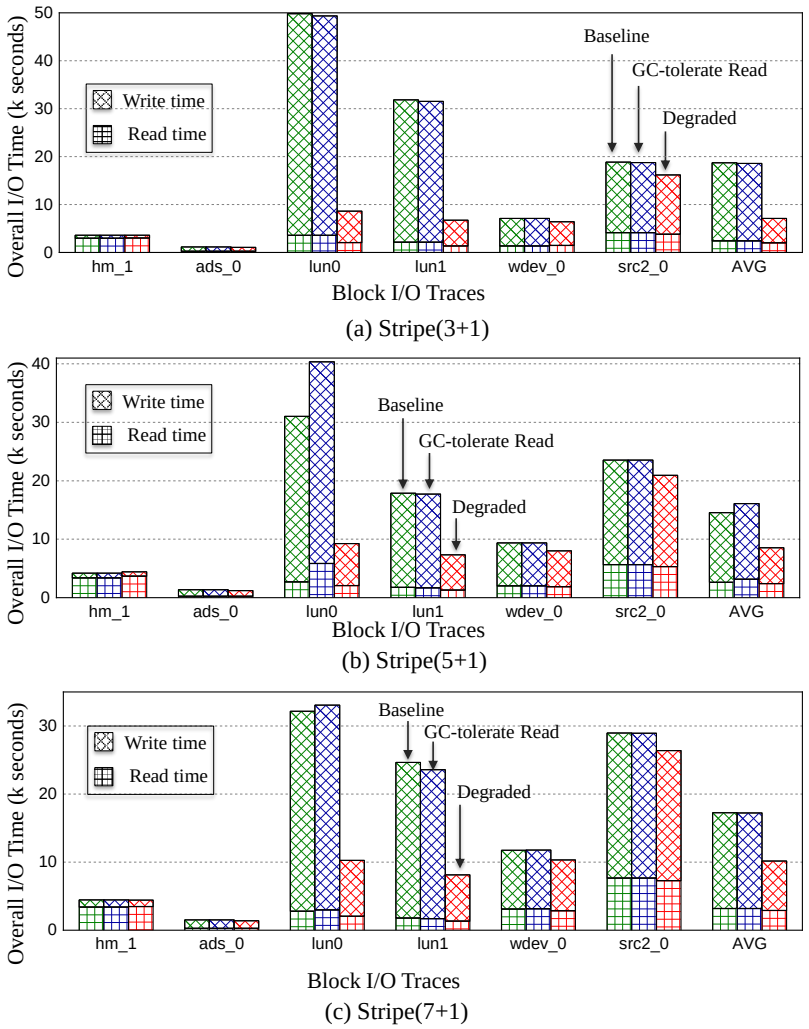


Fig. 13. I/O latency after running the block traces under different size of stripe structure (in configuration of 16 channels).

Experimental results show that our proposal can noticeably cut down the long-tail latency of read requests by 23.1% at the 99.99th percentile, and the overall I/O time by 38.7% on average, in contrast to state-of-the-art methods.

ACKNOWLEDGMENTS

The authors would like to thank anonymous reviewers for their valuable comments. This work was partially supported by “National Natural Science Foundation of China (No. 61872299, No. 62032019)", and “the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2021B06)".

REFERENCES

[1] Shi X., Liu W., and He L., et al. Optimizing the ssd burst buffer by traffic detection. *ACM Transactions on Architecture and Code Optimization (**TACO**)*, 2020.

[2] Kim B., Choi J., and Min S. Design tradeoffs for SSD reliability. In proceedings of *17th USENIX Conference on File and Storage Technologies (**FAST '19**)*, 2019.

[3] Balakrishnan M., Kadav A., and Prabhakaran V. et al. Differential raid: Rethinking raid for ssd reliability. *ACM Transactions on Storage (**TOS**)*, Vol. 6(2): 1-22, 2010.

[4] Aritome S. NAND flash memory technologies. John Wiley & Sons, 2015.

[5] Luo Y., Ghose S., Cai Y., and Erich F. et al. Improving 3D NAND flash memory lifetime by tolerating early retention loss and process variation. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (**POMACS**)*, 2018.

[6] Chan H., Li Y., and Lee P. et al. Elastic Parity Logging for SSD RAID Arrays: Design, Analysis, and Implementation. *IEEE Transactions on Parallel and Distributed Systems (**TPDS**)*, 2018.

[7] Grupp L., Davis J., and Swanson S. The bleak future of NAND flash memory. In proceedings of *10th USENIX Conference on File and Storage Technologies (**FAST '12**)*, 2012.

[8] Xu E., Zheng M., and Qin F. et al. Lessons and actions: What we learned from 10k ssd-related storage system failures. In proceedings of *USENIX Annual Technical Conference (**ATC '19**)*, 2019.

[9] Lee W., and Hong S. et al. 2019. Interpage-Based Endurance-Enhancing Lower State Encoding for MLC and TLC Flash Memory Storages. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, 2019.

[10] Li J., Sha Z., and Cai Z., et al Patch-Based Data Management for Dual-Copy Buffers in RAID-Enabled SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, 2020.

[11] Schroeder B., Merchant A., and Lagisetty R. Reliability of NAND-based SSDs: What field studies tell us. *Proceedings of the IEEE (**PIEEE**)*, 2017.

[12] Kim J., Lee E., and Choi J. et al. Chip-level raid with flexible stripe size and parity placement for enhanced ssd reliability. *IEEE Transactions on Computer (**TC**)*, 2016.

[13] Wang Y., Wang W., and Xie T. et al. CR5M: A mirroring-powered channel-RAID5 architecture for an SSD. In proceedings of 30th Symposium on Mass Storage Systems and Technologies (*MSST '14*), 2014.

[14] Yan S, Li H, and Hao M et al. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. In proceedings of *15th USENIX Conference on File and Storage Technologies (**FAST '17**)*, pp. 15-28, 2017.

[15] DeCandia G, Hastorun D, and Jampani M et al. Dynamo: Amazon's Highly Available Key-value Store. In proceedings of *21st Symposium on Operating Systems Principles (**SOSP '07**)*, 2007.

[16] Google: Taming The Long Latency Tail-When More Machines Equals Worse Results. http://highscalability.com/blog/2012/3/12/googletaming-the-long-latency-tail-when-moremachines-equal.html, 2012.

[17] Misra P, and Borge M, et al. Managing tail latency in datacenter-scale file systems under production constraints. In proceedings of *14th EuroSys Conference (**EuroSys '19**)*, pp. 1-15, 2019.

[18] Wu S, Zhu W, and Liu G et al. GC-aware request steering with improved performance and reliability for SSD-based RAIDs. In proceedings of *2018 IEEE International Parallel and Distributed Processing Symposium (**IPDPS '18**)*, pp. 296-305, 2018.

[19] Wu S, Zhu W, Han Y, et al. GC-Steering: GC-Aware Request Steering and Parallel Reconstruction Optimizations for SSD-Based RAIDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, Vol. 39(12): 4587-4600, 2020.

[20] Pan W, and Xie T. A Mirroring-Assisted Channel-RAID5 SSD for Mobile Applications. *ACM Transactions on Embedded Computing Systems (**TECS**)*, Vol. 17(4): 1-27, 2018.

[21] Cui J, and Liu W et al. Exploiting Disturbance-Aware Read Redirection for Performance Improvement in 3D Flash Memory. In proceedings of *Great Lakes Symposium on VLSI (**GLSVLSI '20**)*, pp. 95-100, 2020.

[22] Zuolo L, Zambelli C, Micheloni R, et al. Memory driven design methodologies for optimal SSD performance. *Inside Solid State Drives (SSDs)*. Springer, Singapore, pp. 181-204, 2018.

[23] Kang Y, Jo Y Y, Cha J, et al. FORESEE: An Effective and Efficient Framework for Estimating the Execution Times of IO Traces on the SSD. *IEEE Transactions on Computers (**TC**)*, 2020.

[24] Silicon Motion Launches Complete 16-Channel PCIe 4.0 NVMe Turnkey Enterprise SSD Controller Solution. https://ir.siliconmotion.com/news-releases/news-release-details/silicon-motion-launches-complete-16-channel-pcie-40-nvme-turnkey/, 2020.

[25] Liang J, Li Y and Chen H et al. Boosting performance of SSD with chip-level RAID by deferring garbage collection. *IEICE Electronics Express(**ELEX**)*, Vol. 15(11):1-12, 2018.

[26] Song W, Zhou Y, and Zhao M et al. EMC: Energy-aware morphable cache design for non-volatile processors. *IEEE Transactions on Computers (**TC**)*, Vol. 68(4): 498-509, 2018.

[27] Hu Y, and Jiang H et al. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. *IEEE Transactions on Computers (**TC**)*, Vol. 62(6): 1141-1155, 2013.

[28] Cui J, Liu J, and Huang J et al. SmartHeating: On the Performance and Lifetime Improvement of Self-Healing SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, Vol. 40(1):52-65, 2020.

[29] Li J, Sha Z, and Cai Z. et al. Patch-based Data Management for Dual-copy Buffers in RAID-enabled SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (**TCAD**)*, Vol. 39(11): 3956-3967, 2020.

[30] Lee C, and Matsuki T et al. Understanding Storage Traffic Characteristics on Enterprise Virtual Desktop Infrastructure. In proceedings of *10th ACM International Systems and Storage Conference (**SYSTOR '17**)*, pp. 1-11, 2017.

[31] Wang S, and Wu F et al. WAS: Wear Aware Superblock Management for Prolonging SSD Lifetime. In proceedings of *56th Annual Design Automation Conference (**DAC '19**)*, pp. 1-6, 2019.

[32] MSRC Traces. http://iotta.snia.org/traces/388.

[33] Microsoft Production Server Traces. http://iotta.snia.org/traces/block-io/158.

[34] Li J, Huang B, and Sha Z. et al. Mitigating Negative Impacts of Read Disturb in SSDs. *ACM Transactions on Design Automation of Electronic Systems (**TODAES**)*, Vol. 26(1), Article No. 3, 24 pages, Oct. 2020.

[35] Tai A., Smolyar I., and Wei, M. et al. Optimizing Storage Performance with Calibrated Interrupts. In Proceedings of USENIX Symposium on Operating Systems Design and Implementation (*OSDI '21*), pp. 129-145, 2021.