Adaptive Selection of Parity Chunk Update Methods in RAID-enabled SSDs

Fan Yang*, Zhigang Cai*✉, Jun Li*, Balazs Gerofi†, Francois Trahay‡
Zhibing Sha*, Mingwang Zhao*, Jianwei Liao*
*College of Computer and Information Science, Southwest University, Chongqing, China
†Intel Corporation, USA; ‡Telecom SudParis, France; ✉Corresponding Author

***Abstract*—RAID-enabled SSDs commonly have unbalanced I/O workloads on their components (e.g., on their SSD channels), and updating parity requires issuing a number of additional read requests (termed as *pre-read*s), which causes long tail latency. To address this issue, we introduce an *adaptive scheme for selecting the update method on parity chunks*. Our scheme uses the most suitable routine for updating parity to better balance I/O accesses over all channels of RAID-enabled SSDs. In particular, it can adaptively select routines including *read-modify-write (RMW)* and *read-construct-write (RCW)*, when renewing data stripes. To this end, we build a mathematical model to assess the time required for updating parity chunks with different update routines by considering the number of *pre-read* requests and the blocked I/O traffic on the affected SSD channels. As a result, it can determine whether the update of parity chunk should be performed with *RMW* or *RCW* during I/O scheduling. Trace-driven experiments illustrate that the proposed scheme can balance the workloads across channels, as well as reduce the long-tail latency of I/O requests by up to `24.8%` at the 99.9th percentile and the overall I/O time by `13.5%` on average, in contrast to existing approaches.**

***Index Terms*—Solid-state Drivers, RAID-5, Load Balance, Update Requests, Read-Modify-Write, Read-Construct-Write**

I. Introduction

NAND flash-based solid-state drives (SSDs) have advantages of fast random access and low energy consumption, so they are widely used as persistent storage in various digital devices [1]–[3]. Benefiting from the rapid development of chip manufacturing technologies, the density of SSDs has increased substantially and the per-unit price of SSDs has consequently decreased [4]–[6]. Modern high density SSD devices, however, are severely impacted by read/write disturb, data retention and low disturbance endurance that directly increase their raw bit error rate (RBER) [7]–[11]. Therefore, efficiently dealing with such noise to ensure the reliability of flash memory-based SSDs has becomes a major challenge [6], [12], [13]. Advanced Error Correction Code (ECC) schemes such as Low Density Parity Check Code (LDPC) can easily cover RBERs at the cost of additional latency due to read retries [14], but they cannot correct chip/channel-level failures in SSDs [15], [16].

In order to ensure the reliability of data, the technology of Redundant Array of Independent Disks (e.g., *RAID-5*) has been introduced into SSDs[1], or called Redundant Array of Independent NAND (*RAIN*) [17], to avoid data loss and data unavailability [6], [12], [16], [18]. Vendors have incorporated RAID technology into SSD products [19], [20] and several research studies have investigated RAID-enabled SSDs to enhance access performance and durability [18], [21], [22]. Specifically, *RAID-5* organizes the data as stripes (labeled as the *N+1* structure), where each stripe consists of `N` data chunks and `1` parity chunk that is XORed with the corresponding data chunks [23], [24]. Lost data chunks can be recovered by referring to the rest of the data chunks and the parity chunk in the given stripe using certain XOR computations [25].

Although enabling *RAID-5* increases SSD reliability, it doubles the number of write operations and requires additional XOR computations while servicing small update requests. This is attributed to the fact that each update to data chunk(s) within a stripe leads to another update on the corresponding parity chunk, referred to as *write penalty* [5], [23], [26]. More specifically, RAID-enabled SSDs generally come with two routines for updating their parity chunks in a data-stripe, the *read-modify-write (**RMW**)* and the *read-construct-write (**RCW**)*. Both update routines need to issue additional read requests (called *pre-reads* [27], [28]) to obtain the data/parity chunks in the given stripe for regenerating the latest version of its data. To be specific, when satisfying the update request(s) and correspondingly renewing the parity chunk of a data stripe, *RMW* requires reading the target data chunk of the update request as well as the old parity chunk. Meanwhile, *RCW* requires reading all data chunks of data stripe except for the one currently being updated. Consequently, the two update methods have different associated costs, i.e., the number of *pre-read* requests, and the I/O scheduler usually makes a choice between the two methods by referring to the number of *pre-read* requests required to update the parity chunk.

RAID-enabled SSDs usually perform *out-place-update* operations and *round-robin* parity placement, that can result in unbalanced I/O across the channels of the SSD device [29]–[31]. Furthermore, unbalanced I/O load generally causes uneven distribution of garbage collection (GC), which further increases I/O congestion on busy SSD channels [32], [33]. The issue is that the critical factor of the congestion status of all RAID components (i.e., SSD channels in the paper) has not been fully considered when selecting the routine for updating a parity chunk. This may greatly impact I/O load evenness over RAID components. As a result, major I/O performance

[1] We use the channel-level *RAID-5* implementation inside SSDs by default in the paper, where a (data/parity) chunk is normally referred to as a page in RAID-enabled SSDs.

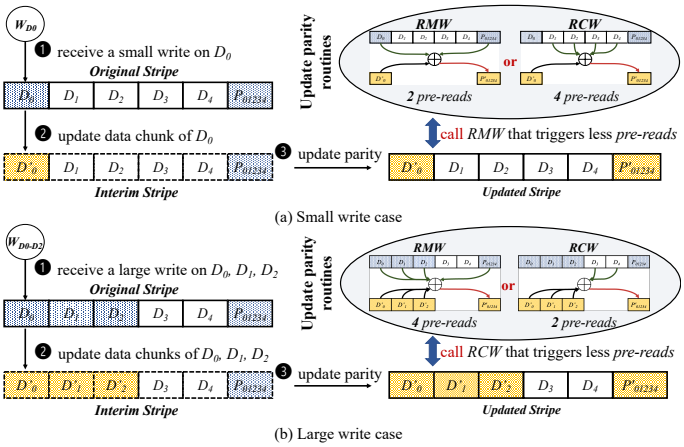


Fig. 1. Updating data stripe in RAID-enabled SSDs with `5+1` stripe configuration. *Note that the interim stripe in the illustration is not in a consistent state*.

metrics, such as I/O response time and long-tail latency will be greatly impacted by the increased number of *pre-read* operations [34].

On the other side, considerable user applications exhibit the feature of update storm, that greatly degrades the storage performance [35], [36]. Specifically, an update storm indicates a large amount of transaction accesses on the same pieces of data in a short interval, thus negatively affecting the quality-of-service (QoS). In particular, Virtual Desktop Infrastructure (VDI) seeks utilizing network connected virtual machines to provide desktop services with easier management, greater availability, and lower cost [37]. As a result, VDI has been widely adopted in offices and universities in recent days. Maintaining high QoS, however, is a difficult problem in the VDI environment due to the high degree of resource sharing. From the perspective of I/O access patterns, VDI applications reveal update burstiness and the largest write throughput in the morning [38].

To adaptively support the selection of update methods for parity chunks, and thus further reduce the tail latency in RAID-enabled SSDs, when running user applications (e.g., VDI applications), we propose a selection method between *RMW* and *RCW* when servicing an update request by referring to not only the number of *pre-read* requests but also to the congestion status of the corresponding SSD channels. In summary, this paper makes the following contributions:

- We introduce an *adaptive selection of update methods for parity chunks* in RAID-enabled SSDs. The proposed scheme intends to avoid worsening I/O workloads on congested RAID components through adaptively selecting the most suitable routine for updating the parity chunk when renewing RAID data stripes.
- We build a mathematical model to assess the time required for updating the parity chunk that considers the factors of the number of *pre-read* requests and the blocked I/O traffics on affected SSD channels. During I/O scheduling, the model helps determine whether the update of a given parity chunk should be performed using *RMW* or *RCW*.
- We perform simulation based experiments by replaying `six` commonly used block traces of real world applications on RAID-enabled SSDs with varied size of stripe configurations. As our measurements indicate, the proposed scheduling approach improves metrics of overall I/O response time, I/O long-tail latency as well as I/O workload balance.

The remainder of this paper is organized as follows: Section II presents the background and motivation of our work. The specifications on the design and implementation of our model are described in Section III. Section IV depicts the evaluation methodology and discusses the results. The related work is summarized in Section V. At last, the paper is concluded in Section VI.

II. Background and Motivation

A. RAID Implementation in SSDs

SSD devices commonly have a functionality of ECC to guarantee data integrity when reading a data page [6], [39]. If errors are detected and the number or span of errors is beyond the ECC capability, the read operation is deemed a failure and the SSD device is notified. In such cases, the RAID technique is employed to regenerate the data and write a new copy onto a free data page of the SSD to maintain RAID guarantees.

As one of standard RAID levels, *RAID-5* consists of block-level striping with distributed parity, and provides advantages in load-balancing and I/O parallelism. RAID-5 has been commonly applied in SSDs either at the chip-level [40] or channel-level, for the purpose of reliability [28]. Figure 1 shows our example of I/O processing on the write request of W_{D0}. For illustration purposes, we use six channels, where each data stripe has five data chunks and one parity chunk. As per *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. Subsequently, the new data chunk and the parity chunk are flushed onto free pages using the same SSD channels.

As seen in Figure 1, the operation first updates the data chunk of D_0 to D'_0 after completing the write request of W_{D0}. Next, it takes advantage of *RMW* instead of *RCW* (since the former routine triggers less *pre-read*s), to renew the parity chunk to P'_{01234} so that the stripe is kept consistent. Note that the updated data chunks or the parity chunk should be placed on the same channel(s) of the SSD to keep the stripe structure.

B. Routines of Updating Parity

To ensure the consistency of data stripe while servicing an update (write) request, there are two methods to update the parity chunk of a data stripe, *RMW* and *RCW* [26]. This section describes both methods in detail, also by referring to Figure 1.

In the *RMW* routine, the updated parity chunk can be obtained by *XORing* the obsolete parity chunk with the obsolete data chunk(s) and the updated data chunk(s) that are related

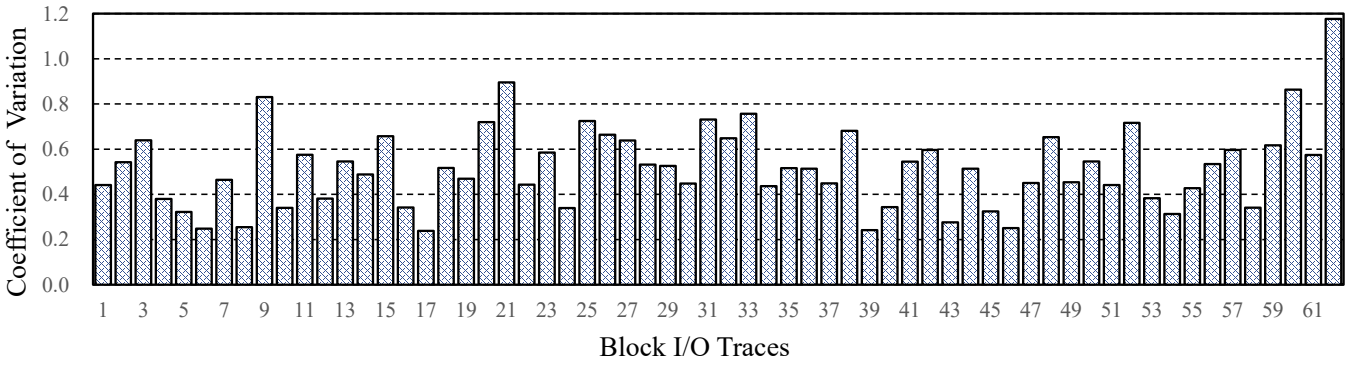


Fig. 2. The values of coefficient of variation of the number of blocked requests in RAID components after replaying the traces from the LUN block collection [38]. The number in the X-axis is the sequential number of the trace in the collection folder of *systor17-additional-01*.

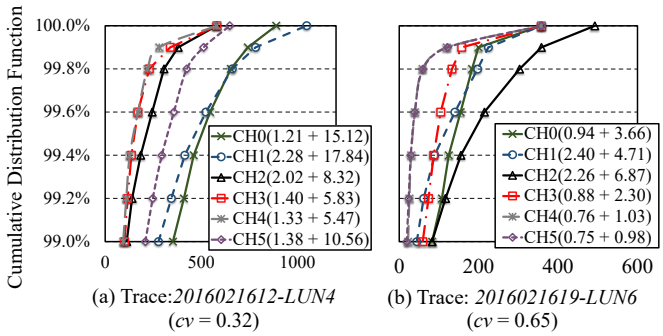


Fig. 3. CDF of I/O latency on the most congested channel and the remaining channels of the same stripes in conventional RAID-enabled SSDs (unit: ms).

to the write request(s). As the example shows in Figure 1(a), the operation triggers 2 *pre-read* requests on the parity chunk of P_{01234} and the obsolete data chunk of D_0. Note that the up-to-date contents of D'_0 are buffered in the DRAM cache associated with the write request of W_{D0} and consequently it does not require a *pre-read*. Finally, the operation carries out an XOR computation to obtain the latest contents of parity chunk of P'_{01234} for keeping the stripe consistent.

In the *RCW* routine, the updated parity chunk can be obtained by *XORing* the updated data chunk(s) that are related to the write request(s) with unchanged data chunks in the same stripe. As seen in Figure 1(a), the operation triggers 4 *pre-read* requests on the data chunks of D_1, D_2, D_3, and D_4, and then carries out an XOR computation to renew the parity chunk. Conventional SSD controllers generally select either *RMW* or *RCW* for updating the parity chunk of data stripe by comparing the number of *pre-read* requests required by each routine [28], [41], [42]. According to the general approach, *RMW* is selected to update the parity chunk for small write requests, as seen in Figure 1(a), and *RCW* is used for updating the parity chunk of larger write requests that span multiple data chunks [26], [43], as seen in Figure 1(b).

C. Motivation

In order to disclose the channels of RAID-enabled SSDs exhibit different levels of I/O congestion when running user applications, we specifically analyzed the *LUN* block I/O traces [38] and recorded the number of blocked I/O requests as the indicator of I/O congestion after replaying them. In fact, the *LUN* block trace collection is recently gathered from a part of an enterprise virtual desktop infrastructure[2].

Subsequently, we performed χ^2 hypothesis testing whether the number of blocked requests on channels conformed to a uniform distribution or not for each trace, and we found that a majority of tests rejected the uniform distribution at $P<0.001$ [44], see **Appendix A**. This fact implies that imbalanced I/O workloads are common in real-world applications, such as those run in *VDI* environments.

Furthermore, we computed the value of coefficient of variation (*cv*) for the number of blocked requests in all channels to quantify the degree of imbalance workloads over all SSD channels. A larger value of *cv* implies that all channels have varied congestion status, whereas a smaller value of *cv* indicates that all channels have a similar congestion level [45]. In fact, *cv* is defined as the ratio of standard deviation to the mean of the blocked requests on the SSD channels, and we obtain the value of *cv* after replaying a benchmark with the following steps: ❶ we record the number of the enqueued I/O requests in each channel, after a request is serviced, and then sum the numbers of blocked I/O requests once the benchmark is finished. ❷ we divide the sum of blocked I/O requests by the total number of completed I/O requests, to yield the average number of blocked I/O requests in each channel. ❸ we can calculate the standard deviation of blocked I/O requests across all channels. ❹ we can compute the value of *cv* through dividing the standard deviation with the mean of blocked I/O requests in all channels.

Figure 2 shows the results of the *cv* values for the used traces with a stripe configuration of 5+1. As seen, the *cv*

[2]The *LUN* trace collection consists of 438 pieces of traces and organizes them as 7 folders. We present the results of all traces in the first folder of *systor17-additional-01* in the paper.

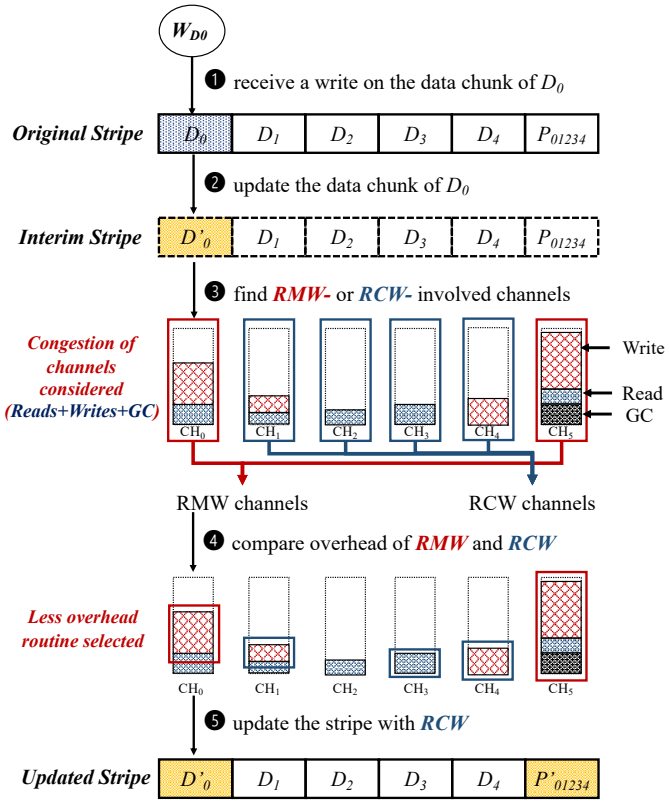


Fig. 4. The high level overview of adaptive selection of updating on parity chunks. *The details of computing the overhead of updating routines in Step ❹ can be found in Figure 5* of Section III-B.

value is distributed between `0.24∼1.18`, which indicates that all traces capture imbalanced I/O workloads across all RAID components.

Modern SSDs are composed by a controller, and multiple flash channels. When the applications sends an I/O request, the SSD device picks it up and inserts it into the (per-channel) I/O queue for processing. In order to boost I/O performance, the SSD controller manages the channels to enable accessing their data in a parallel manner. Since the macro request may span several pages, SSD device will parse such request into several page-sized sub-requests and mapping them to the target channels, and the request is considered to be completed only when all sub-requests have been serviced. That is to say, the response time of I/O request may become larger unexpectedly if its sub-requests are mapped to the most congested channels, causing tail latency [46]. In fact, it has been verified that imbalanced I/O workloads will exacerbate the problem of tail latency, and the most congested channels commonly have the worst responsiveness in RAID-enabled SSDs [1], [41]. To further illustrate this fact in our example scenario of *VDI* applications, we present the results of long tail latency of I/O requests in two randomly selected *LUN* traces. Figure 3 reports the results as Cumulative Distribution Function.

In the figure, X-axis labels correspond to I/O latency in milliseconds, and Y-axis labels scale from the `99.0`th to the `100.0`th percentile latency. The average numbers of blocked requests (reads+writes) on channels are shown in the legend, and more blocked requests on a SSD channel commonly represent a congested I/O workload on it. To obtain the number of blocked requests, we recorded the number of the enqueued requests in each channel after a request is serviced. When the benchmark was finished, we averaged the number of blocked requests. We conclude that the worst tail latency comes from the most congested channels, and such observations drive us to introduce dynamic selection of update methods for parity chunks by considering not only the number of *pre-read*s, but also the congestion status of affected RAID channels.

III. Adaptive Selection of Parity Update Methods

A. Architectural Overview

The basic idea of our proposal is to select an update routine on parity chunks by considering both factors of the number of *pre-read*s and the congestion level of the I/O workloads across the channels participating in a given stripe in RAID-enabled SSDs. To this end, we build a cost assessment model (see Section III-B) for evaluating the queuing overhead of updating routines of *RMW* and *RCW*, and recommend using the routine with less overhead for updating the parity chunk. As a result, our approach yields a more even I/O workload distribution across SSD channels and ensures lower tail latencies for the I/O requests while running the applications.

Figure 4 presents a high-level overview of completing a write request of W_{D0} and the relevant update on the parity chunk by using our proposal. As seen, after updating the data chunk of D_0, it evaluates the update cost of *RMW* and *RCW* by using the proposed assessment model. Subsequently, it chooses the *RCW* routine that has less overhead to update the parity chunk for ensuring the consistency of the stripe, even though it needs to issue more *pre-read* requests.

B. Cost Assessment Model

We construct a mathematical model for assessing the time required for using both routines to update the parity chunk of the data stripes. Table I summarizes the symbols and their definitions used in the model.

Assuming that the stripe structure consists of K channels, labelling as CH_0, CH_1, ..., CH_{K-1}. The number of read and write requests in these channels can be represented as R_0, R_1,..., R_{K-1} and W_0, W_1, ..., W_{K-1}. In addition to processing read and write requests, SSD channels endures time-consuming tasks of GC operations, we employ T_{GC} representing the GC latency. The parameter Θ is an indicator for the set of involved channels corresponding to a given data stripe. Specifically, $\Theta = \{ 0, 1, ..., K-1 \}$. We also define the notations of Θ_{RMW} and Θ_{RCW} that mean the channel sets while using the routines of *RMW* and *RCW* to update a given parity chunk, respectively. Thus, the following two equations hold:

TABLE I
NOTATION DESCRIPTIONS

Symbol	Explanation
t_R	Read latency
t_W	Write latency
t_E	Erase latency
CH_i	The ith channel of SSD
W_i	The number of write requests on CH_i
R_i	The number of read requests on CH_i
T_{GC}	Average GC latency
Θ	The channel set of data stripe
Θ_{RMW}	The involved channel set with RMW
Θ_{RCW}	The involved channel set with RCW
T_{RMW_Update}	The update latency with RMW
T_{RMW_Delay}	The delay on enqueued req. with RMW
T_{RMW}	The overall latency of RMW
T_{RCW_Update}	The update latency caused by RCW
T_{RCW_Delay}	The delay on enqueued req. with RCW
T_{RCW}	The overall latency of RCW
$Cntr_{Move}$	The number of page movements
$Cntr_{ER}$	The number of erasure

$$\Theta_{RMW} \cup \Theta_{RCW} = \Theta \tag{1}$$

$$\Theta_{RMW} \cap \Theta_{RCW} = \varnothing \tag{2}$$

Furthermore, the GC process blocks normal I/O requests aiming at the target channel, and its latency consists of the time required for moving valid pages on the GC block to another free block (i.e., *page moves*) and the time needed for erasing the GC block. More exactly, we estimate the time overhead of the future GC operation by using two per-chip counters of $Cntr_{Move}$ and $Cntr_{ER}$. Specifically, $Cntr_{Move}$ and $Cntr_{ER}$ respectively represent the total number of page movements and erases during garbage collection, by referring to [46]. The erasure time is usually a fixed value, t_E. Thus, the T_{GC} can be calculated by Equation 3:

$$T_{GC} = (t_R + t_W) \times (Cntr_{Move}/Cntr_{ER}) + t_E \tag{3}$$

Both update routines need to create a write operation on the channel corresponding to the original parity chunk, so that our model does not take the overhead of the write operation into account. Considering that read operations exhibit faster performance than write operations, and users demonstrate a heightened sensitivity to delays in data retrieval, SSDs usually assign a higher priority to read requests [47], [48]. When a specific stripe update routine is selected, a number of *pre-read* requests must be issued at the end of read requests but the beginning of the write requests in the I/O queue of the affected SSD channels. The successful completion of these *pre-read* requests is crucial because the XOR computation for the parity chunk depends on completing all associated *pre-reads*. In other words, the overall pace of this process is typically determined by the slowest *pre-read* request among them. Moreover, we

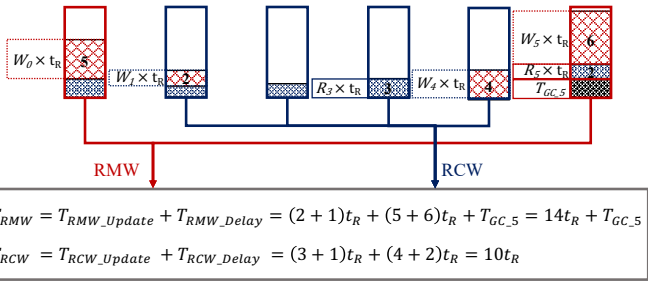


Fig. 5. Assessment example of update overhead of ***RMW*** and ***RCW*** by using the proposed model. *Note that the original write operation was completed on CH_0 and the parity update operation should be on CH_5.*

understand that all I/O requests on the GC channel cannot be serviced until the GC operation is completed. Then, the time cost of updating the parity chunk with different routines can be defined as following two parts.

The first part of wait time in the time cost after using two routines to update the parity chunk is defined as T_{RMW_Update} and T_{RCW_Update} with Equations 4 and 5. In which, T_{GC_i} represents the current GC time on the *ith* channel, and is set as 0 while no GC on the SSD channel.

$$T_{RMW\_Update} = \max_{i \in \Theta_{RMW}} \{t_R \cdot (R_i + 1) + T_{GC\_i}\} \tag{4}$$

$$T_{RCW\_Update} = \max_{i \in \Theta_{RCW}} \{t_R \cdot (R_i + 1) + T_{GC\_i}\} \tag{5}$$

The second part of the time cost is the delay on the subsequent write requests in the I/O queue caused by the additional *pre-read* request. In fact, as read operations are accorded superior priority, SSDs categorize unprocessed read requests and write requests into two separate queues. That is to say, the *pre-read* request is inserted to the tail of the queue of read requests, resulting in each queued write request within the same channel incurring an additional waiting period, corresponding to the processing time of one read request. Then, we define T_{RMW_Delay} and T_{RCW_Delay} in Equations 6 and 7, to represent the total delay time caused by the inserted *pre-read* requests with two update routines.

$$T_{RMW\_Delay} = t_R \cdot \sum_{i \in \Theta_{RMW}} W_i \tag{6}$$

$$T_{RCW\_Delay} = t_R \cdot \sum_{i \in \Theta_{RCW}} W_i \tag{7}$$

At last, we obtain the total time cost of T_{RMW} and T_{RCW} by summing up the aforementioned two parts of time overhead, as defined in Equations 8 and 9.

$$\begin{aligned} T_{RMW} &= T_{RMW\_Update} + T_{RMW\_Delay} \\ &= \max_{i \in \Theta_{RMW}} \{t_R \cdot (R_i + 1) + T_{GC\_i}\} + t_R \cdot \sum_{i \in \Theta_{RMW}} W_i \end{aligned} \tag{8}$$

$$T_{RCW} = T_{RCW\_Update} + T_{RCW\_Delay} = \max_{i \in \Theta_{RCW}} \{t_R \cdot (R_i + 1) + T_{GC\_i}\} + t_R \cdot \sum_{i \in \Theta_{RCW}} W_i \quad (9)$$

For a given write request, the model determines the values of T_{RMW} and T_{RCW} by observing the current status of I/O workloads and the GC state on the relevant RAID components, and selects the preferred routine that has smaller overall latency updating the parity chunk. To better illustrate the specifications of our proposed model, Figure 5 gives an assessment example of the update overheads with both routines to end the write operation of W_{D0} (which was originally shown in Figure 4).

On the one hand, the value of T_{RMW_Update} is related to the number of read requests and the GC time on CH_5 since it has the largest overall latency of processing GC and read requests in the RMW-channel set. The value of T_{RMW_Delay} is related to the sum of the number of write requests in the I/O queues of CH_0 and CH_5. On the other hand, T_{RCW_Update} only depends on the number of read requests in the I/O queue of CH_3, and T_{RCW_Delay} is relevant to the sum of the number of write requests in the I/O queues of CH_1 to CH_4.

As a result, the *RCW* method will be selected for ending the update on the parity chunk based on the output of our model, despite the fact that it has to issue more *pre-read*s.

C. Implementation Details

Algorithm 1 shows the implementation specifics on how an update routine is selected for a given parity chunk. Lines `7` and `8` locate the involved channel set using the update routines of *RMW* or *RCW*. Afterwards, it computes the total overhead of *RMW* (Lines `9 - 14`) and *RCW* (Lines `15 - 18`) for updating the parity chunk. At last, it chooses the method with less overall time cost to finish the update operation on the parity chunk.

Algorithm 1: Adaptive Selection of Update on Parity

Input: args of t_R, Req and its stripe structure;
Output: null;

```
1  /*Initializing the overhead of RMW and RCW*/
2  T_RMW_Update = T_RCW_Update = 0;
3  T_RMW_Delay = T_RCW_Delay = 0;
4  T_RMW = T_RCW = 0;
5  if Req is write on an existing stripe then
6      /*Find the channels which selected by routines*/
7      Θ_RMW = find_channel(RMW, Req);
8      Θ_RCW = find_channel(RCW, Req);
9      /*Compute overhead of RMW*/
10     T_RMW_Update = max { t_R · (R_i in Θ_RMW + 1) + T_GC_i in Θ_RMW };
11     /*∑W_i is the sum of writes in queues of Θ_RMW*/
12     T_RMW_Delay = t_R · ∑W_i;
13     /*Sum two parts overhead*/
14     T_RMW = T_RMW_Update + T_RMW_Delay;
15     /*Compute overhead of RCW*/
16     T_RCW_Update = max { t_R · (R_i in Θ_RCW + 1) + T_GC_i in Θ_RCW };
17     T_RCW_Delay = t_R · ∑W_i;
18     T_RCW = T_RCW_Update + T_RCW_Delay;
19     if T_RMW > T_RCW then
20         update_stripe(RCW, Req);
21     else
22         update_stripe(RMW, Req);
```

IV. Experimental Evaluation

This section first describes the experimental settings and then presents the results to validate the feasibility of the proposed selection of update routine on RAID parity. Finally, we analyze the time and space overhead of our proposal.

A. Environmental Setup

Due to its advances in a diverse set of configurations and its validation accuracy against a real hardware platform, the *SSDsim* simulator [50] has been widely used in many SSD focused studies [51]. Thus, we performed trace-driven simulations using *SSDsim* (ver 2.1) to evaluate the newly proposed scheme. In order to further characterize the effectiveness of our proposal using different RAID stripe structures, we test data stripes using both `5+1` and `7+1` configurations. Table II presents the settings of SSDsim in our experiments. Since we employ LDPC to correct bit errors, the read latency directly depends on the level of LDPC soft decision. The basic read time is configured as `0.045ms`, and the read time is increased by `0.024ms` per read retry after upgrading a LDPC level [52]. Then, the read time spans from `0.045ms` to `0.819ms`, reflecting the levels of LDPC soft decisions. Within our mathematical model, the read time of t_R is adjusted in response to the respective LDPC levels.

TABLE II
Experimental settings of *SSDsim*

Parameters	Values	Parameters	Values
Channel Size	6/8	*Read latency*	0.045-0.819ms
Chip Size	4	*Write latency*	0.7ms
Plane Size	4	*Erase latency*	3.5ms
Block per plane	512	*XOR latency*	0.019ms
Page per block	64	*GC threshold*	10%
Page size	8KB	*RAID level*	5
FTL scheme	Page	*ECC*	7-level LDPC
Wear-leveling	Static	*Stripe Struct.*	5+1 or 7+1

We employed `six` widely used block I/O traces, covering a wide range of write ratios, as workloads for the RAID-5 system to verify the effectiveness of our proposal in various application scenarios. Among them, ***hm_0*** comes from the *MSR Cambridge* block I/O collection [53]. Three block I/O traces are captured from real world VDI ap-

TABLE III
SPECIFICATIONS ON SELECTED TRACES (ORDERED BY THE *cv*)

Traces	# of Req.	Wr Ratio	Wr Size	Upd R	*cv*
hm_0	3,993,316	64.5%	8.3KB	97.8%	0.10
lun0	2,387,992	14.1%	27.7KB	79.3%	0.32
lun1	824,068	45.4%	11.2KB	79.8%	0.60
lun2	948,330	35.6%	11.9KB	81.1%	0.65
Ali_0	623,524	86.8%	13.4KB	85.8%	0.78
Ali_1	398,009	90.8%	12.7KB	87.4%	1.24

TABLE IV
CUMULATIVE DISTRIBUTION FUNCTION OF REQUEST INTERVALS IN THE SELECTED TRACES

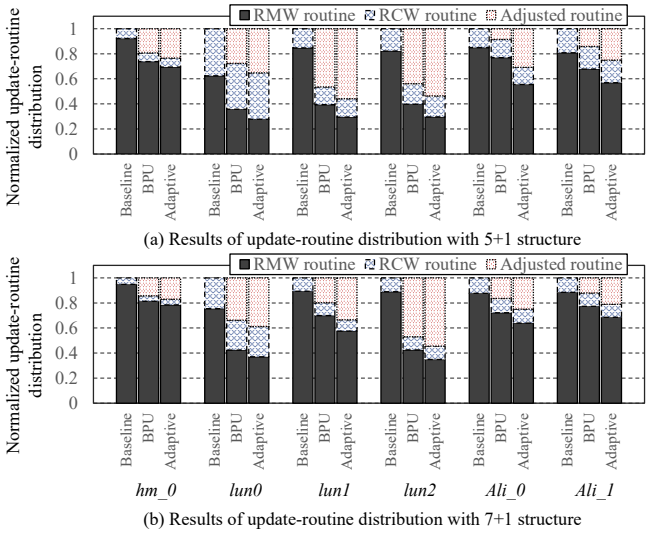
Traces	20%	40%	60%	80%	100%
hm_0	0.1us	0.3us	1.9us	0.2ms	0.9s
lun0	39.9us	385.8us	879.1us	2.0ms	11.4ms
lun1	31.0us	397.1us	836.9us	4.2ms	2.0ms
lun2	32.3us	433.9us	888.1us	4.0ms	2.9ms
Ali_0	1.0us	1.3us	1.5us	2.7us	9.9ms
Ali_1	1.9us	2.2us	2.6us	4.9us	200.0ms

plications [38], which are *additional-01-2016021612-LUN4* (labeled as ***lun0***), *additional-01-2016021618-LUN6* (labeled as ***lun1***) and *additional-01-2016021619-LUN6* (labeled as ***lun2***). Moreover, we employed two recent traces from *Alibaba Cloud* [54], corresponding to twelve-hour trace segments of a 40 GB virtual disk. The size is notably align to the capacity of our emulated SSD device, labeled as ***Ali_0*** and ***Ali_1***. All these selected benchmarks are also commonly used in the domain of SSD optimization [31], [55], [56].

The detailed information about these I/O traces are reported in Tables III and IV. It should be noted that the metric of **Upd R** in Table III is the ratio of update (write) requests to all write requests in the trace. Table IV presents the details on the intervals between two I/O requests in the selected trace, in Cumulative Distribution Function, to reveal the bursty level of I/O requests. Generally, a small value of interval means congested I/O workloads, and will cause a large average I/O latency for servicing a read/write request.

Besides our proposal (labeled as ***Adaptive***), we included the following two schemes for comparison in our evaluation:

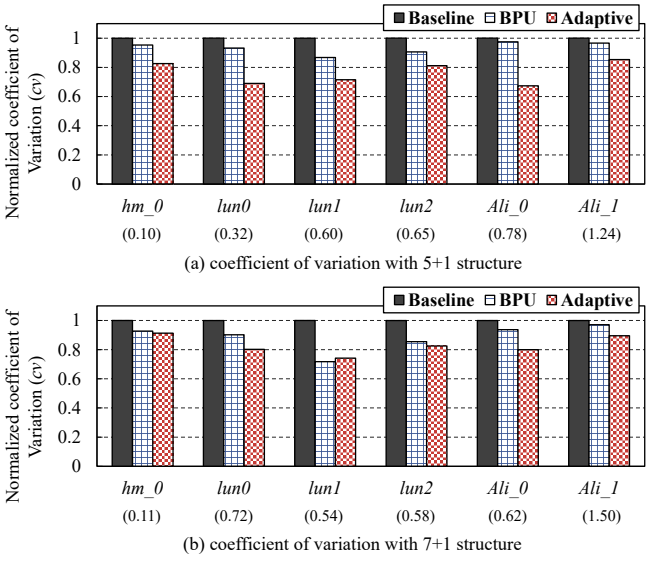
- ***Baseline*** [28], [42], is the common update policy for updating the parity chunk in RAID-enabled SSDs. It chooses the routine (i.e., *RMW* or *RCW*) that has fewer *pre-read* requests when updating a given stripe. Note that this policy does not consider the status of the I/O workloads on the affected SSD channels.
- ***BPU*** [27], which is a balanced parity update algorithm considering the factor of the congestion level of I/O queues. This algorithm empirically classifies all updates on parity into several categories, on the basis of the number of *pre-read*s and the total length of I/O queues. After that, it employs the recommended routine to update parity chunks in each category.
  We argue that *BPU* is the most related work to our proposal, which takes the factor of channel congestion into consideration for selecting the routine to update the parity chunk. The *BPU* scheme, however, lacks adaptivity and universality, as it employs certain fixed **empirical thresholds** for guiding the selection of update routine and fails to consider the difference in the impact on enqueued read/write requests and GC operations caused by the inserted *pre-read* request.



(a) Results of update-routine distribution with 5+1 structure

(b) Results of update-routine distribution with 7+1 structure

Fig. 6. Normalized update-routine distribution after using selected comparison method.



(a) coefficient of variation with 5+1 structure

(b) coefficient of variation with 7+1 structure

Fig. 7. Normalized coefficient of variation (*cv*) after using selected comparison methods. *Note that the numbers underlying X-axis are the absolute values with Baseline.*

B. Results and Discussion

To measure the validity of our proposal, we utilize the following three metrics in our experiments: ***(a)*** *the congestion level of SSD channels,* ***(b)*** *I/O response time, and* ***(c)*** *long-tail latency.*

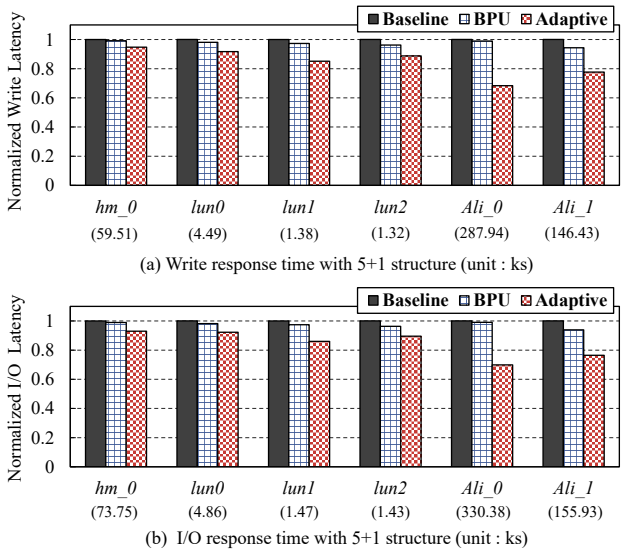


Fig. 8. Results of I/O performance metrics of write latency (a) and overall I/O latency (b) with `5+1` stripe structure.

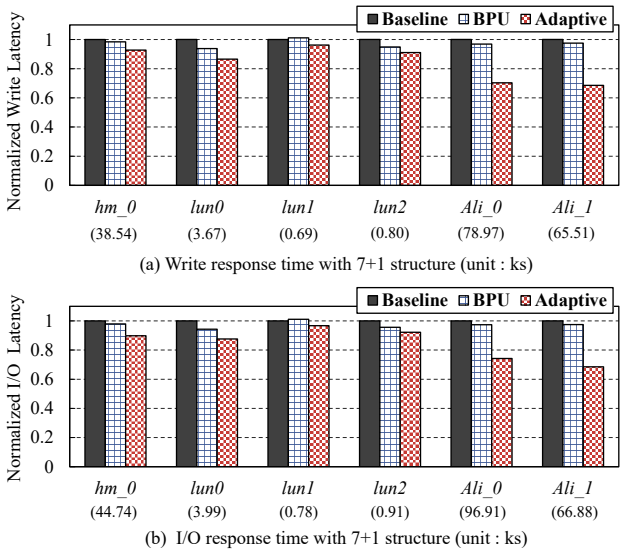


Fig. 9. Results of I/O performance metrics of write latency (a) and overall I/O latency (b) with `7+1` stripe structure.

1) Update Routine Selection and Congestion Analysis: We record the distribution of varied update routines for updating parity, when processing the selected benchmarks with *Baseline*, *BPU* and *Adaptive*. Figure 6 shows the results.

On the one hand, *Baseline* makes use of either *RMW*- or *RCW*-based updates on parity chunks, according to their number of *pre-read* requests to service the update on the parity chunks. On the other hand, *BPU* and *Adaptive* will select the routine even though it has more *pre-read*s, by considering the I/O workload status. Thus, they have one more part of *Adjusted* in the update routine selection to indicate using the opposite recommended by *Baseline*.

Compared to *BPU*, the proposed *Adaptive* scheme brings about more *Adjusted* operations by `56.2%` on average. This illustrates the fact that our proposal causes more updates on the parity chunks with the adjusted routine by considering the factor of real-time workload congestion so that it can yield more I/O improvements in contrast to *BPU*.

The indicator of coefficient of variation (*cv*) of blocked I/O requests in the I/O queues of SSD channels reflects the status of load balance over all channels. Figures 7(a) and 7(b) demonstrate the *cv* results with `5+1` and `7+1` stripe structure, respectively. As shown, both the *BPU* and *Adaptive* schemes contribute to a notable reduction of *cv* of blocked I/O requests in RAID components as they consider the current congestion status of I/O workloads when selecting the routine to update parity. More importantly, compared with the related work of *BPU*, our proposal reduces the value of *cv* by `12.1%` on average. This observation verifies the fact that our proposal contributes to the I/O workload balance of applications by timely selecting an appropriate update routine with the support of the cost assessment model.

2) I/O Response Time: I/O response time is the most important metric to reflect the performance of SSD devices. We have thus measured the time required for replaying the block I/O traces by using the different methods for updating parity. Figures 8 and 9 present the normalized results of overall I/O response time that consists of the read time and the write time for SSDs with `5+1` and `7+1` stripe structures, respectively. It is worth mentioning that all schemes cause high overall I/O latencies for replaying the traces from MSR Cambridge and Alibaba Cloud, by comparing to running the LUN traces. We argue that these traces are write-intensive, and have very small values of the interval between two requests in a major part of workloads (refer to Table IV), implying congested I/O workloads in them, so that they requires more time for servicing all requests.

In contrast to *Baseline*, both *BPU* and *Adaptive* noticeably reduces the overall I/O latency of all traces, as *Baseline* does not take the factor of real-time I/O workloads into account when selecting the routine for updating parity chunks even though a part of channels are heavily congested with I/O requests. As reported in Figures 6(a) and 6(b), both optimized approaches do service a part of updates on parity chunks with the routine that may have more *pre-read*s, to purposely refrain from worsening the congestion status on some SSD channels.

Specifically, *Adaptive* improves the overall I/O latency by up to `29.8%` in all traces, compared to *BPU*. This is because *BPU* employs certain pre-trained, empirical thresholds to identify the current situation which it uses to select the corresponding routine (i.e., *RMW* or *RCW*) for updating parity. On the other hand, the overhead assessment model of *Adaptive* can benefit timely selecting the preferred update routine, by referring to both the number of *pre-read*s and the workloads of stripe-involved channels into account, without any pre-defined thresholds. Thus, *Adaptive* can more accurately avoid dispatching *pre-read*s onto the most congested channels, which relieves the I/O workloads on them and thus improves I/O

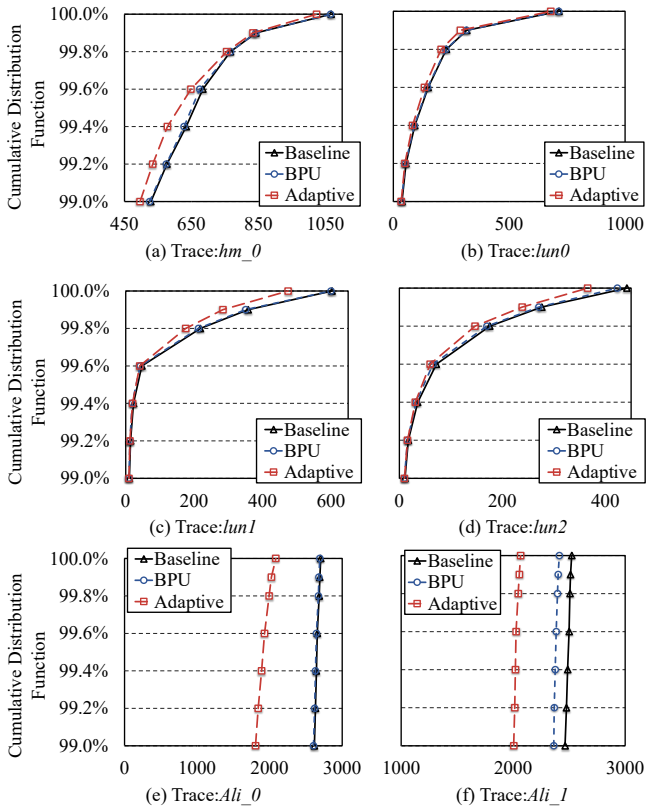


Fig. 10. Results of long tail latency after replaying the selected traces, with `5+1` stripe structure (unit : ms).

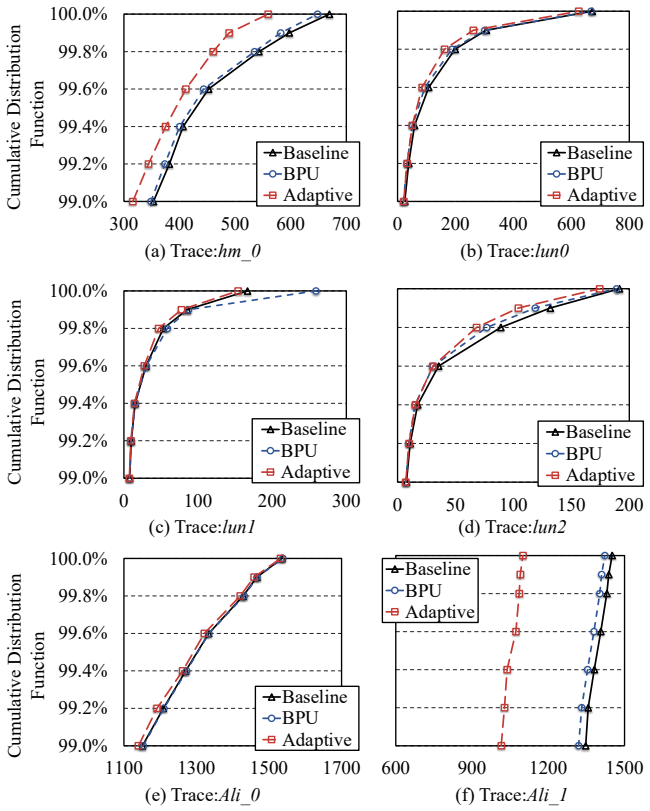


Fig. 11. Results of long tail latency after replaying the selected traces, with `7+1` stripe structure (unit : ms).

performance.

Furthermore, we observe that the I/O performance improvement induced by *BPU* is limited when replaying the traces from MSR Cambridge and Alibaba Cloud, which is in accordance with the values of *cv* in Figure 7. We argue this is because these traces will generate burst I/Os and *BPU* is particularly ineffective to deal with such I/Os. On the other side, our proposal shows outstanding capability for the application having intensive I/O workloads that are unevenly distributed over all RAID components. In summary, the I/O improvements brought about by *BPU* are tightly sensitive to the pre-trained parameters and the specific I/O workload of application. To the contrary, our *Adaptive* 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 in a real-time manner.

3) Long-tail Latency: As discussed in our motivations, imbalanced I/O workloads on RAID-enabled SSD channels can postpone some I/O requests and further aggravate the tail latency issue. To reduce the long-tail latency is another target of the proposed scheme, Figures 10 and 11 show the comparison of long-tail latency (in Cumulative Distribution Function) for requests after replaying the selected traces with different size of stripe structure.

The lines of *Baseline* are almost the lowest ones since it does not make use of any optimization strategies to cut down the long-tail latency. Our proposed *Adaptive* approach exhibits better long-tail latency than that using other selected schemes. More exactly, *Adaptive* can significantly reduce the long-tail latency by `14.6%`, and `12.9%` on average at the 99.9th percentile, compared to *Baseline* and *BPU*. This fact proves that adaptively selecting the routine to update parity by using the proposed cost assessment model, can efficiently minimize the impacts on normal I/O requests caused by the inserted *pre-read* requests, and thus ensure I/O responsiveness.

To better confirm our proposed method can balance I/O workloads among all RAID components after replaying the traces, we recorded the tail latency of each channel in the RAID-enabled SSDs, and calculated the average, minimum, and maximum tail latency of all channels. Figure 12 and 13 present the normalized result of tail latency of all channels, with the `5+1` stripe structure and the `7+1` stripe structure, respectively. As seen, the height of the bars represents the average long-tail latency, and the error bars indicate the maximum and minimum values respectively. Since *Baseline* does not take the workload balance into consideration, the average tail latency is the largest in the most cases. On the other side, we can see our proposal of *Adaptive* outperforms the other two comparison schemes in the majority of cases, with the best average tail latency and the smallest difference between the maximum and minimum tail latency of all channels. This fact proves our method can better achieve workload balance among

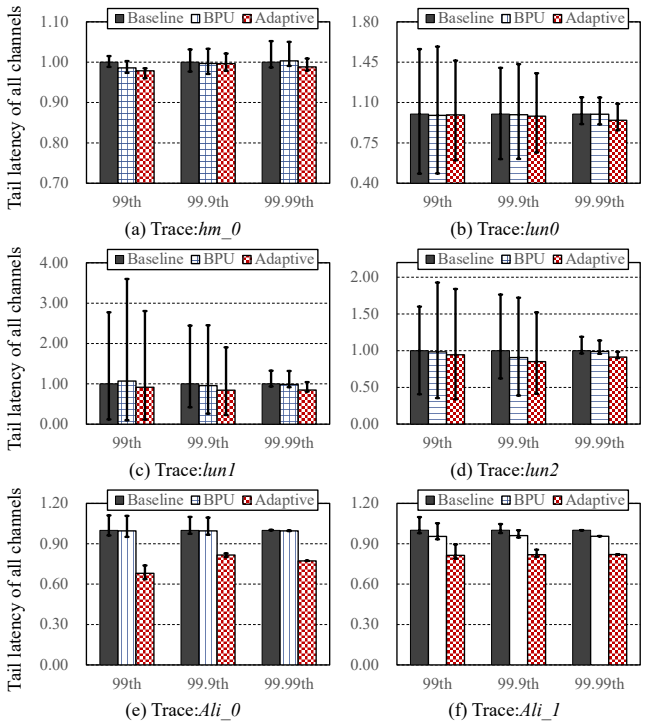


Fig. 12. Results of long tail latency among all channels after replaying the selected traces, with 5+1 stripe structure.

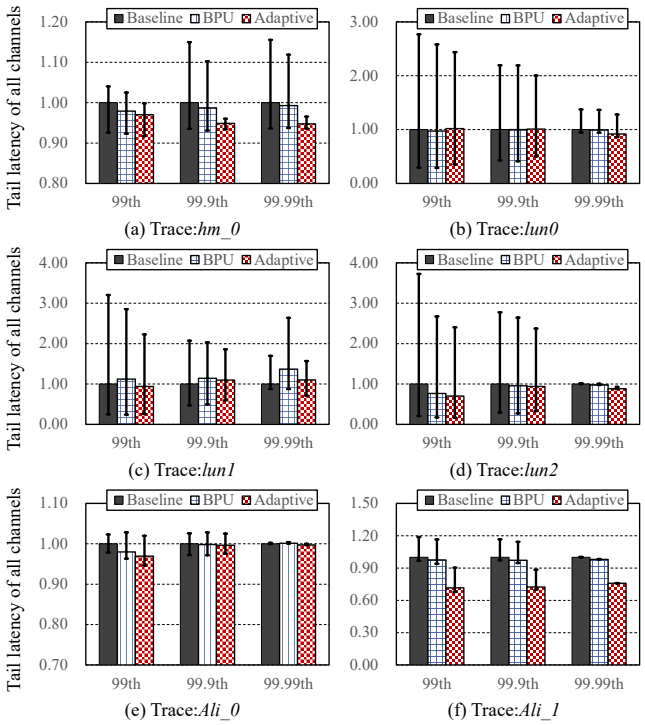


Fig. 13. Results of long tail latency among all channels after replaying the selected traces, with 7+1 stripe structure.

all channels in the RAID-enable SSD, and then contribute to better I/O performance.

Another interesting observation from Figures 12 and 13 is that, *Adaptive* does not outperform *Baseline* and *BPU* after replaying the LUN traces, on the measure of average tail latencies across all channels. This is because these traces generally have large intervals between two I/O requests and will not lead to heavily congested channels, directly limiting the improvement room of our method. We emphasize that, but, our proposal of *Adaptive* can yield the least difference of tail latency of all channels after replaying the most of traces, indicating the best workload balance among all RAID components.

C. Overhead

The main memory overhead of our proposal is due to the additional storage required for the parameters used by the cost assessment model. The model has to record the number of enqueued read and write requests of each channel, which translates to 64B = 8 channels $\times$ (2 counts $\times$ 4B).

On other other hand, *BPU* records not only the number of enqueued I/O requests of each channel, but also some parameters and thresholds to identify the current state of SSD for guiding the selection of update routine on the parity chunk.

In summary, the space overhead caused by our proposal accounts for a very small part of the storage capacity of SSD, resulting in an acceptable amount of memory space in SSDs.

With respect to time overhead, the proposed approach only requires to additionally compute the overhead of different

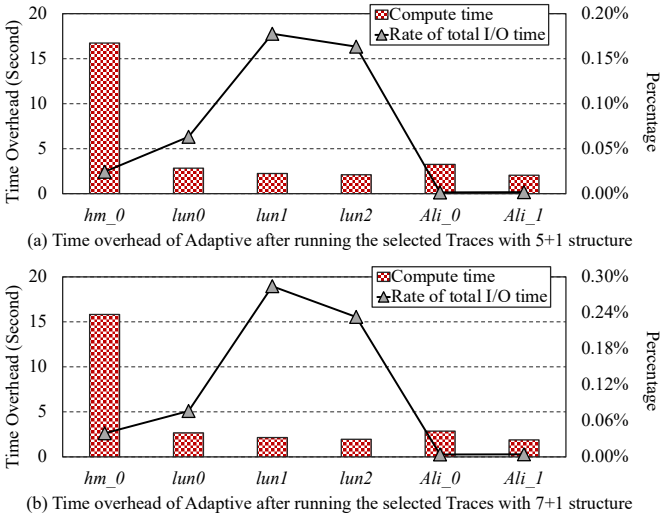


Fig. 14. Time overhead of *Adaptive* after running the selected Traces.

update routines (i.e., T_{RMW} and T_{RCW} in the cost assessment model). To measure the compute overhead of *Adaptive*, we run the selected traces on an ARM-based platform, simulating the main controller of SSD device has a limited compute processing capacity. As the results shown in Figure 14, *Adaptive* causes time overhead between 1.9 and 16.7 seconds, accounting for an average of 5.7μs per update parity operation, or less than 0.3% of the overall I/O time. Then, we consider that the time overhead caused by the computing

is acceptable, even though our experiments are done on the ARM Cortex A7 Dual-Core CPU with 800MHz and 128MB of memory.

V. Related Work

When a write request modifies part of data stripe, the parity chunk needs to be updated by using either *RMW* or *RCW*. Both update schemes require issuing a number of *pre-read* requests onto relevant RAID components for the purpose. In conventional SSD RAID implementations, such as *CR5M* [28], [41] and *SWO* [42], the basic selection rule is to compare the number of *pre-read*s of *RMW* with that of *RCW*, and the routine having the less number of *pre-read*s will be employed to update the parity chunk.

In general, *RMW* is suitable for small write requests, and *RCW* fits the large write requests. Sevilla et al. [43] proposed modifying the write selection algorithm to classify medium-writes as small-writes and then to knowingly update the parity with *RCW*, based on their experimental observations that show *RCW* can contribute to better system performance in borderline size of requests.

Sun et al. [57] and Thomasian [58] successively proposed their methods for selecting either *RMW* or *RCW* when updating the data chunks, to specifically enhance the overall performance of RAID systems that are built on the top of hard disk drives. On the other hand, Chen et al. [27] focused on the load balance in SSDs and proposed selecting the update routine by referring to the current I/O workloads. Specifically, their approach uses certain **empirical and pre-trained** thresholds to identify the current situation, according to the number of *pre-read*s and the length of I/O queues in the RAID system. We argue that it is better to have a common model to timely direct the selection of update routines on parity, for eventually yielding I/O workload balance among all SSD channels by considering not only whether channels are busy or not, but also the levels of busyness of involved SSD channels caused by serving normal I/O requests.

Besides, Jiang et al. [59] have proposed *FusionRAID* targeting small write requests, to avoid directly updating the data stripe. *FusionRAID* temporally replicates data chunks of small write requests to lighten I/O congestion, and later converts the replicated chunks into RAID stripes according to various configurable thresholds.

VI. Conclusion

This paper proposes adaptive routine selection for updating parity chunks of data stripes in RAID-enabled SSDs. Our goal is to achieve a balanced I/O workload distribution and to guarantee their I/O responsiveness. To this end, we first construct a mathematical cost assessment model to estimate the overhead of two update routines by referring to the number of *pre-read*s corresponding to the update on the parity chunk, as well as the blocked I/O traffics on the different RAID components. Then, we determine how the update on the parity should be serviced with the routine introducing less overhead by choosing between *RMW* or *RCW*.

Experimental results show that our proposal substantially decreases the long-tail latency of I/O requests, by up to `24.8%` at the 99.9th percentile, as well as the overall I/O time by more than `13.5%` on average, in contrast to state-of-the-art methods.

Acknowledgment

This work was partially supported by "National Natural Science Foundation of China (No. 62032019)", "Natural Science Foundation Project of CQ CSTC (No. 2022NSCQ-MSX0789)" and "The Chongqing Graduate Research and Innovation Project (No. CYS23205)". Dr. Zhigang Cai is the corresponding author.

References

[1] Sha Z, Li J, and Song L, et al. Low I/O Intensity-aware Partial GC Scheduling to Reduce Long-tail Latency in SSDs. *ACM Transactions on Architecture and Code Optimization(**TACO**)*, 18(4): 1-25, 2021.

[2] Lin H, Li J, Sha Z, et al. Adaptive Management with Request Granularity for DRAM Cache inside NAND-based SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 2022.

[3] Kang M, Lee W, Kim J, et al. PR-SSD: Maximizing Partial Read Potential by Exploiting Compression and Channel-Level Parallelism. *IEEE Transactions on Computers(**TC**)*, 72(3): 772-785, 2022.

[4] Zhang W, Cao Q, and Jiang H, et al. Improving overall performance of TLC SSD by exploiting dissimilarity of flash pages. *IEEE Transactions on Parallel and Distributed Systems(**TPDS**)*, 31(2): 332-346, 2019.

[5] Tang C, Wan J, and Zhu Y, et al. RAFS: A RAID-Aware File System to Reduce the Parity Update Overhead for SSD RAID. In proceedings of *2019 Design, Automation & Test in Europe Conference & Exhibition(**DATE**)*, 1373-1378, 2019.

[6] Kim B, Choi J, and Min S. Design tradeoffs for SSD reliability. In proceedings of *17th USENIX Conference on File and Storage Technologies(**FAST**)*, 281-294, 2019.

[7] 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**)*, 26(1): 1-24, 2020.

[8] Jaffer S, Mahdaviani K, Schroeder B. Improving the Reliability of Next Generation SSDs using WOM-v Codes. *20th USENIX Conference on File and Storage Technologies(**FAST'22**)*, 117-132, 2022.

[9] Zhao M, Li J, and Cai Z, et al. Block Attribute-aware Data Reallocation to Alleviate Read Disturb in SSDs. In proceedings of *the 24th International Conference on Design, Automation, and Test in Europe(**DATE '21**)*, 1096-1099, 2021.

[10] Cui J, Zeng Z, Huang J, et al. Improving 3-D NAND SSD Read Performance by Parallelizing Read-Retry. IEEE *Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 42(3): 768-780, 2022.

[11] Zhang Y, Hao W, Niu B, et al. Multi-view Feature-based SSD Failure Prediction: What, When, and Why. In proceedings of *21st USENIX Conference on File and Storage Technologies(**FAST'23**)*, 409-424, 2023.

[12] Kishani M, Ahmadian S, and Asadi H. A modeling framework for reliability of erasure codes in SSD arrays. *IEEE Transactions on Computers(**TC**)*, 69(5): 649-665, 2019.

[13] Kim J, Lee J, and Choi J, et al. Improving SSD reliability with RAID via elastic striping and anywhere parity. In proceedings of *43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks(**DSN**)*, 1-12, 2013.

[14] Zhao K, Zhao W, and Sun H, et al. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In proceedings of *11th USENIX Conference on File and Storage Technologies(**FAST '13**)*, 243-256, 2013.

[15] Jaffer S, Maneas S, Hwang A, et al. The reliability of modern file systems in the face of SSD errors. *ACM Transactions on Storage(**TOS**)*, 16(1): 1-28, 2020.

[16] Wang S, Wu F, and Lu Z, et al. Ward: Wear aware raid design within ssds. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 37(11): 2918-2928, 2018.

[17] NAND Flash Media Management Through RAIN. Retrieved from https://www.micron.com/-/media/client/global/documents/products/technical-marketing-brief/brief_ssd_rain.pdf

[18] Li J, Gerofi B, Trahay F, et al. Rep-RAID: An Integrated Approach to Optimizing Data Replication and Garbage Collection in RAID-Enabled SSDs. In proceedings of *International Conference on Languages, Compilers and Tools for Embedded Systems(**LCTES**)*, 99-110,2023.

[19] P320h 2.5-Inch PCIe NAND SSD Features. Retrieved from https://www.micron.com/-/media/client/global/documents/products/data-sheet/ssd/p320h_2_5.pdf

[20] Huawei Tecal ES3000 Application Accelerator Review. Retrieved from https://www.storagereview.com/review/huawei-tecal-es3000-application-accelerator-review

[21] Sha Z, Li J, Cai Z, et al. Degraded mode-benefited I/O scheduling to ensure I/O responsiveness in RAID-enabled SSDs. *ACM Transactions on Design Automation of Electronic Systems(TODAES)*, 27(6): 1-24, 2022.

[22] Kim J, Lee E, Choi J, et al. Chip-level RAID with flexible stripe size and parity placement for enhanced SSD reliability. *IEEE Transactions on Computers (**TC**)*, 65(4): 1116-1130, 2014.

[23] Chung C, Hsu H. Partial parity cache and data cache management method to improve the performance of an SSD-based RAID. *IEEE Transactions on Very Large Scale Integration(VLSI) Systems(**TVLSI**)*, 22(7): 1470-1480, 2013.

[24] Im S, Shin D. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Transactions on Computers(**TC**)*, 60(1): 80-92, 2010.

[25] Hong D, Ha K, Ko M, et al. Reparo: A Fast RAID Recovery Scheme for Ultra-large SSDs. *ACM Transactions on Storage(**TOS**)*, 17(3): 1-24, 2021.

[26] Xu G, Tan Z, Feng D, et al. Cap: Exploiting Data Correlations to Improve the Performance and Endurance of SSD RAID. In proceedings of *IEEE 36th International Conference on Computer Design(**ICCD**)*, 59-66, 2018.

[27] Chen Y, Xu Y, and Li Y, et al. Balanced Parity Update Algorithm with Queueing Length Awareness for RAID Arrays. In proceedings of *2016 IEEE 22nd International Conference on Parallel and Distributed Systems(**ICPADS**)*, 818-825, 2016.

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

[29] 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 *IEEE International Parallel and Distributed Processing Symposium(**IPDPS**)*, 296-305, 2018.

[30] Yadgar G, Gabel M, Jaffer S, et al. Ssd-based workload characteristics and their performance implications. *ACM Transactions on Storage(**TOS**)*, 17(1): 1-26, 2021.

[31] Liu R, Liu D, Chen X, et al. Self-Adapting Channel Allocation for Multiple Tenants Sharing SSD Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 41(2): 294-305, 2021.

[32] Yan S, Li H, and Hao M, et al. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage(**TOS**)*, 13(3): 1-26, 2017.

[33] Li Y, Shen B, Pan Y, et al. Workload-aware elastic striping with hot data identification for SSD RAID arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 36(5): 815-828, 2016.

[34] Kim S, Bae J, Jang H, et al. Practical Erase Suspension for Modern Low-latency SSDs. In proceedings of *2019 USENIX Annual Technical Conference(**ATC**)*, 813-820, 2019.

[35] Kavalanekar S, Worthington B, Zhang Q, et al. Characterization of storage workload traces from production windows servers. In proceedings of *2008 IEEE International Symposium on Workload Characterization(**ISWC**)*, 119-128, 2008.

[36] Narayanan D, Donnelly A, Thereska E, et al. Everest: Scaling Down Peak Loads Through I/O Off-Loading. In proceedings of *8th USENIX Symposium on Operating Systems Design and Implementation(**OSDI**)*, 15-28, 2008.

[37] Shamma M, Meyer D, and Wires J, et al. Capo: Recapitulating storage for virtual desktops. In proceedings of *USENIX Conference on File and Storage Technologies(**FAST**)*, 2011.

[38] Lee C, and Matsuki T, et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In proceedings of *the 10th ACM International Systems and Storage Conference(**SYSTOR**)*, 1-11, 2017.

[39] Lee W, and Hong S, et al. 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**)*, 27(9): 2033-2045, 2019.

[40] 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**)*, 29(10): 2241-2253, 2018.

[41] 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**)*, 1-10, 2014.

[42] Mei L, Feng D, and Zeng L, et al. A stripe-oriented write performance optimization for RAID-structured storage systems. In proceedings of *2016 IEEE International Conference on Networking, Architecture and Storage(**NAS**)*, 1-10, 2016.

[43] Sevilla M, Wacha R, and Brandt S. RAID4S-modthresh: Modifying the write selection algorithm to classify medium-writes as small-writes. *Technical Report (No. UCSC-SOE-12-10)*, University of California, USA.

[44] Bearden W, Sharma S, and Teel J. Sample size effects on chi square and other statistics used in evaluating causal models. *Journal of marketing research*, 19(4):425-430, 1982.

[45] Wu C, and He X. GSR: A global stripe-based redistribution approach to accelerate RAID-5 scaling. In proceedings of *International Conference on Parallel Processing(**ICPP**)*, 460-469, 2012.

[46] Elyasi N, Arjomand M, and Sivasubramaniam A et al., Exploiting intra-request slack to improve SSD performance. In proceedings of *he Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (**ASPLOS**)*, 375-388, 2012.

[47] Lv Y, Shi L, Li Q, et al. Access characteristic guided partition for read performance improvement on solid state drives. *2020 57th ACM/IEEE Design Automation Conference (**DAC**)*, 1-6, 2020.

[48] Gao C, Shi L, Zhao M, et al. Exploiting parallelism in I/O scheduling for access conflict minimization in flash-based solid state drives. *2014 30th Symposium on Mass Storage Systems and Technologies (**MSST**)*, 1-11, 2014.

[49] Wu G, Huang P, He X. Reducing SSD access latency via NAND flash program and erase suspension. *Journal of Systems Architecture(**JSA**)*, 60(4): 345-356, 2014.

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

[51] Zhou Y, Wu F, and Huang W, et al. LiveSSD: A low-interference RAID scheme for hardware virtualized SSDs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems(**TCAD**)*, 40(7): 1354-1366, 2020.

[52] Du Y, Zhou Y, Zhang M, et al. Adapting Layer RBERs Variations of 3D Flash Memories via Multi-granularity Progressive LDPC Reading. In proceedings of *Design Automation Conference (**DAC**)*, 1-6, 2019.

[53] Narayanan D, Thereska E, and Donnelly A et al. Migrating server storage to SSDs: analysis of tradeoffs. In proceedings of *The European Conference on Computer Systems (**EuroSys**)*, 145-158, 2009.

[54] Alibaba Block Traces. Alibaba Group. Retrieved from https://github.com/alibaba/block-traces

[55] Wu J, Cai Z, Yang F, et al. Polling sanitization to balance I/O latency and data security of high-density SSDs[J]. *ACM Transactions on Storage (**TOS**)*, 2024.

[56] Gao C, Ye M, and Li Q et al. Constructing Large, Durable and Fast SSD System via Reprogramming 3D TLC Flash Memory. In proceedings of *The IEEE/ACM International Symposium on Microarchitecture(**MICRO**)*, 493-505, 2019.

[57] Sun D, Xu Y, Li Y, et al. Efficient parity update for scaling RAID-like storage systems. In proceedings of *IEEE international conference on networking, architecture and storage(**NAS**)*, 1-10, 2016.

[58] Thomasian A. Reconstruct versus read-modify writes in RAID. Information processing letters, 93(4): 163-168, 2005.

[59] Jiang T, Zhang G, and Huang Z et al. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In proceedings of *USENIX Conference on File and Storage Technologies(**FAST**)*, 355-370, 2021.

APPENDIX

Table A.2 reports the results of χ^2 hypothesis testing on *LUN* traces, in which the trace name is the sequential number of trace in the collection folder of *systor17-additional-01*. All tests rejected the uniform distribution at P<0.001, implying imbalanced I/O workloads are common in the *LUN* trace collection.

TABLE A.2
RESULTS OF χ^2 HYPOTHESIS TESTING ON *LUN* TRACES.

Trace	**1#**	**2#**	**3#**	**4#**	**5#**	**6#**
χ^2	854,484,849	79,719,420	109,985,422	49,749,672	27,003,878	9,184,267
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**7#**	**8#**	**9#**	**10#**	**11#**	**12#**
χ^2	31,310,151	14,978,171	346,083,789	24,382,869	62,889,713	33,891,180
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**13#**	**14#**	**15#**	**16#**	**17#**	**18#**
χ^2	4,619,822	4,229,208	46,708,063	4,097,840	1,539,935	5,803,152
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**19#**	**20#**	**21#**	**22#**	**23#**	**24#**
χ^2	9,478,078	3,661,880	17,526,959	2,631,865	2,625,367	21,642,197
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**25#**	**26#**	**27#**	**28#**	**29#**	**30#**
χ^2	9,779,624	6,027,253	5,320,520	3,278,062	2,722,826	25,132,491
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**31#**	**32#**	**33#**	**34#**	**35#**	**36#**
χ^2	6,228,995	2,645,198	4,442,138	3,679,142	3,502,246	4,958,340
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**37#**	**38#**	**39#**	**40#**	**41#**	**42#**
χ^2	7,617,855	7,456,892	1,342,894	2,202,148	2,776,262	8,215,202
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**43#**	**44#**	**45#**	**46#**	**47#**	**48#**
χ^2	4,171,076	11,591,926	3,418,844	2,635,552	2,500,641	9,272,870
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**49#**	**50#**	**51#**	**52#**	**53#**	**54#**
χ^2	4,607,131	55,315,372	4,373,151	18,210,202	2,802,453	1,722,316
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**55#**	**56#**	**57#**	**58#**	**59#**	**60#**
χ^2	15,822	43,831	217,441	5,565	3,986	80,538
P value	<0.001	<0.001	<0.001	<0.001	<0.001	<0.001
Trace	**61#**	**62#**				
χ^2	30,323,188	11,741,528				
P value	<0.001	<0.001				