Cocktail: Mixing Data With Different Characteristics to Reduce Read Reclaims for NAND Flash Memory

Genxiong Zhang, Yuhui Deng, Yi Zhou, Shujie Pang, Jianhui Yue, and Yifeng Zhu

***Abstract*—A large number of read-disturb-induced rewrites are performed in the background [also known as *Read Reclaim* (RR)] to alleviate the read-disturb issue in NAND flash memory-based SSDs. RR can significantly degrade the performance and shorten the service life of SSD in read-intensive workloads. To address this issue, we propose a novel read-disturb management approach called *Cocktail* that mixes a small proportion of hot-read pages with a large proportion of cold-read pages, thereby avoiding clustering hot-read pages into a few blocks. Motivated by the insight that RR operations are frequently triggered by hot read-pages, Cocktail first prefills a portion of each block with cold data extracted from user requests. Then, Cocktail fills the prefilled blocks with write-back data caused by RR to create read-balanced blocks. We integrate two thresholds, write pool capacity and the ratio of RR-write data to User-write data, into Cocktail to govern the ratio of write-back data caused by RR to data of user requests in a block. Cocktail dynamically adjusts the two thresholds according to the characteristics of RR. Cocktail is conducive to decentralizing hot write-back data caused by RR across a broad range of blocks, thereby reducing the occurrence of second-time RR and the number of overall block reads. We compare Cocktail with three existing schemes baseline, redFTL, and IPR in terms of SSD service life, SSD response time, write amplification, and the number of garbage collections (GCs) under ten real-world workload conditions. Experimental results show that compared with the existing schemes, Cocktail reduces the number of RRs, the average response time, the 99-percentile tail latency, and the number of GCs by an average of 40.77%, 10.82%, 5.40%, and 12.29%, respectively. Cocktail also alleviates the write amplification of the three alternative schemes by an average of 49.57%.**

Manuscript received 30 March 2022; revised 13 July 2022 and 9 September 2022; accepted 11 October 2022. Date of publication 14 October 2022; date of current version 20 June 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 62072214; in part by the Guangdong Basic and Applied Basic Research Foundation under Grant 2021B1515120048; and in part by the Open Project Program of Wuhan National Laboratory for Optoelectronics under Grant 2020WNLOKF006. This article was recommended by Associate Editor C. Bolchini. *(Corresponding author: Yuhui Deng.)*

Genxiong Zhang and Shujie Pang are with the Department of Computer Science, Jinan University, Guangzhou 510632, China (e-mail: zgx787839110@163.com; p_shujie@163.com).

Yuhui Deng is with the Department of Computer Science, Jinan University, Guangzhou 510632, China, and also with the Wuhan National Laboratory for Optoelectronics, Wuhan 430079, China (e-mail: tyhdeng@jnu.edu.cn).

Yi Zhou is with the TSYS School of Computer Science, Columbus State University, Columbus, GA 31097 USA (e-mail: zhou_yi@columbusstate.edu).

Jianhui Yue is with the Department of Computer Science, Michigan Technological University, Houghton, MI 49931 USA (e-mail: jyue@mtu.edu).

Yifeng Zhu is with the Department of Electrical and Computer Engineering, The University of Maine, Orono, ME 04469 USA (e-mail: zhu@eece.maine.edu).

Digital Object Identifier 10.1109/TCAD.2022.3214679

***Index Terms*—3-D NAND flash, data storage systems, flash translation layer, latency, read disturb.**

I. Introduction

3-D NAND flash memory is now widely used as a storage device. Read-disturb noise has become a major reliability concern for high-density 3-D NAND flash memory. Traditional read-disturb management techniques are inadequate to mitigate the read-disturb issue. To address this problem, we focus on reducing the number of overall block reads and the number of write-backs caused by second-time read reclaims (RRs). We develop a read-disturb management approach called *Cocktail* that leverages a decentralized distribution strategy to place hot write-back data of RRs across a wide range of blocks. Such a decentralized distribution strategy facilitates balancing read access to data blocks, thereby reducing the total number of block reads and the occurrence of second-time RR. At the heart of Cocktail are write frontiers and write pool, which collaboratively combines cold data with hot data into read-balanced blocks. We show how a write frontier extracts and prefills cold data from user requests into blocks. We illustrate how Cocktail leverages a write pool to eventually fill hot data of read-reclaim-induced write-backs into prefilled blocks to create read-balanced blocks.

The following three motivations make Cocktail desirable and achievable.

1) There exists a pressing need for untangling the read-disturb problem in 3-D NAND flash-based SSD devices.
2) RR for mitigating read-disturb significantly degrades SSD performance.
3) Read-disturb management techniques designed to address the issues caused by RR are inadequate for modern SSDs.

The main cause of the read-disturb problem is read accumulation. When reading a page of a block, NAND flash memory must apply a pass-through voltage to all other pages in the same block, which acts like a weak programming operation. Such an operation seriously disturbs a block after the block is read too many times. NAND flash memory has to repeat read operation until the data of a disturbed page is correctly retrieved [1]. As a result, it takes several times longer for NAND flash memory to read a page when read-disturb occurs [2]. An even worse case is that when no correct data are obtained after multiple reads for a disturbed page, NAND

flash memory will permanently lose the disturbed page, which should be definitely avoided.

RR imposes a significant impact on response time and service lifetime of modern SSDs [3]. RR is a common approach used to mitigate the read-disturb issue in 3-D NAND flash [3]. Similar to garbage collection (GC) [4], if there is valid data in a disturbed block, the valid data must be written to a new free block (i.e., read-reclaim-induced write-back), followed by an erase operation on the disturbed block. A disturbed block can be fully recovered after it is erased. However, under read-intensive workloads, SSDs must trigger a large number of erase operations to eliminate possible read errors. These erase operations will extend response time, increase write amplification, and shorten service lifetime of SSDs. In particular, if a normal I/O request conflicts with an RR operation, the normal I/O request has to yield and may be postponed for a long time. Such a response time delay can seriously degrade quality-of-service of I/O intensive applications [5], [6]. Moreover, the undesired erase operations triggered in RR can remarkably shorten the lifespan of NAND flash-based SSDs due to the maximum erase count per SSD block.

A number of read-disturb management approaches have been proposed to address the issues caused by RR [7], [8], [9]. Many existing approaches share a simple strategy: write the data of a disturbed block back to the same block after RR. Unfortunately, when using this simple strategy, hot disturbed blocks may remain in the hot state, which will trigger a second-time RR. In contrast to this simple write-back strategy, some other studies distribute RR data across different pages, aiming to avoid centralizing all hot read data into a small number of blocks [8]. However, because this simple decentralized strategy is not aware of data popularity, hot data is very likely to aggregate into hot blocks again, triggering second-time RRs. Unlike the aforementioned read-disturb management solutions, our decentralized strategy dynamically adjusts the proportion of hot data in a block according to data popularity. Our observations (see Section IV) show that most of the write-back data caused by read-reclaim is hot read data. Therefore, it is feasible to extract cold data from user write requests, place them in a block proportionally in advance, and reserve the rest of the block for future hot write-back data filling. Following these design principles, Cocktail leverages a write pool to quickly combine hot write-back data into prefilled blocks when RR write-back data arrives, thereby preventing second-time RR and creating read-balanced blocks.

In summary, we make the following three contributions on improving the reliability and performance of SSDs.

1) We analyze the root cause of read-disturb errors and the adverse impacts of RR on SSDs, which help us to pinpoint the limitations and shortcomings of previous read-disturb management methods.
2) We propose a low-cost and high-efficiency read-disturb management solution—*Cocktail*—to improve the reliability and performance of high-density 3-D NAND. Cocktail applies a decentralized write-back strategy to balance read accesses to blocks, thereby reducing the number of RRs and the number of block reads.

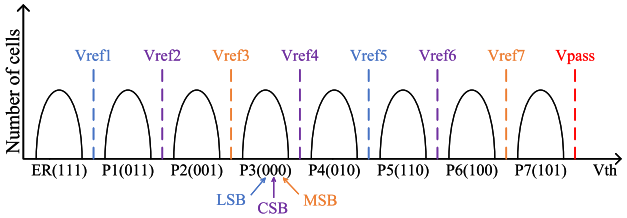


Fig. 1. Voltage distribution.

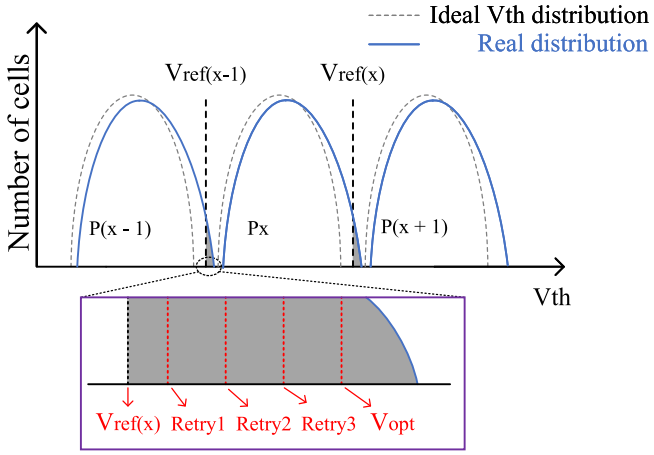


Fig. 2. Voltage shift and read retry.

3) We validate Cocktail with ten real-world workloads. Experimental results show that our Cocktail is far superior to three alternative management approaches in terms of the number of block erase operations, read response time, and the number of GCs.

The remainder of this article is organized as follows. The background and motivation are discussed in Sections II and III. The system design and implementation of Cocktail are present in Section IV. Section V validates the performance of Cocktail by extensive experiments. Section VI summarizes the related work. Finally, Section VII shows conclusions including the contributions of this research.

II. Background

A. Threshold Voltage Range

3-D NAND flash memory cell stores data in the form of threshold voltages. Fig. 1 shows an example cell of eight regions separated by seven reference voltages (i.e., Vref1–Vref7). Vpass is the pass-through voltage serving as the upper bound of the threshold voltage [10]. The voltage region that the flash cell falls into indicates the cell's current state. States "111," "011," "001," "000," "010," "110," "100," and "101" represent logical data "0," "1," "2," "3," "4," "5," "6," and "7," respectively.

Ideally, a NAND flash cell performs a read operation by applying one or more read reference voltages to the word-line (WL) (see Section II-B) containing the data to be read. However, in the real-world environment, cell voltage distribution can be shifted by various circuit-level noises (see Fig. 2). Specifically, a large portion of flash cells can be read

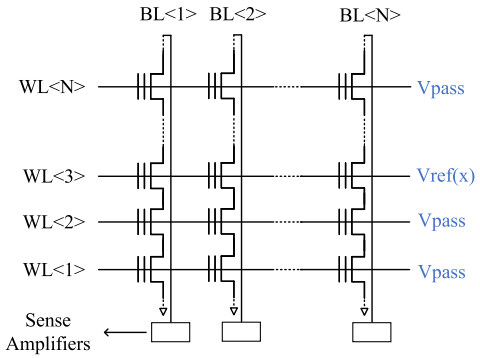


Fig. 3. NAND flash block structure.

incorrectly if flash memory applies a preset reference voltage to WLs. Although such read errors can be corrected by an error-correcting code (ECC) engine, a failed read operation is unavoidable when a read error exceeds the correction range of the ECC engine. To handle a failed read operation, flash memory adjusts its read reference voltage to perform another read operation to read the same page. Such an operation is called a *read retry* [1], which is several times longer than a normal read and seriously affects read performance.

B. Read-Disturb Errors

Modern flash memory is suffering from read-disturb errors. Fig. 3 depicts a typical NAND flash block structure, where the control gates of the flash memory cells in the same vertical position connect to the same WL, whereas the cells in the same column form a bit line (BL).

Modern flash memory strives to pass through unread cells by ensuring the pass-through voltage is higher than the read reference voltage, aiming to reduce errors during read operations. For example, to read a single cell on a BL, flash memory must apply a pass-through voltage to the WL of the read cell so that all other cells on the same BL are not read [11]. Although the pass-through voltage is lower than the programming voltage that moves the cell's threshold voltage to a desired range, it still has a "weak programming" effect. This read-disturb effect inadvertently shifts threshold voltages, causing unbearable read delays and data loss.

C. Read Reclaim

3-D NAND flash memory relies heavily on RR to address the read-disturb problem. The key idea of RR is to rewrite the valid data of a block to a new block and erase the block if the block is suffered from the read-disturb problem.

RR is triggered when the number of reads for a block exceeds a preset threshold—the maximum number of tolerable reads for a block. Since RR needs to rewrite the data of a disturbed block to a new block, it generates a large number of writeback data, which remarkably affects SSD response time. Another issue is that erase operations caused by RR will significantly shorten the SSD lifetime due to NAND's limited erase count. Therefore, reducing RR count becomes a pressing need in modern SSDs.

D. Write Amplification

There are two main problems with SSDs, namely, write amplification and read disturbance. Solutions to these two problems often require co-existence and effective integration.

In order to alleviate the read disturbance problem, it is necessary to avoid placing too much hot read data in the same block. However, this strategy is inapplicable to alleviate the write amplification problem. Specifically, because SSD uses an out-of-place update strategy [12], hot-write data should be put together, so that the effective pages in the victim block will be reduced during GC. A widely used strategy to solve the write amplification problem is to collect hot write data in one place and cold write data in another [13], [14], [15]. In this study, we employ write frontier [15] to write data sequentially into physical blocks, which can be found in Section IV that details how we effectively integrate Cocktail with this write amplification mitigation approach.

III. Motivation

To understand data popularity in real-world workloads, we characterize ten traces under the same conditions (see Section V for a detailed description of our infrastructure and methodology). Fig. 4 shows the frequency of requests and their data popularity distribution. Requests with the same request id have the same logical address and request size. Frequency represents the number of occurrences of requests with the same request id. There are some requests whose frequency exceeds the maximum value of the y-axis, and these requests are represented as maximum values to make the hierarchy more visible in the graph. We take the maximum value as the standard. We mark requests with a frequency less than half of the maximum value in blue, and requests with a frequency greater than half in red. Fig. 4 demonstrates that data with relatively low popularity is more than that with relatively high popularity in different traces.

To further examine the workloads characteristics, we conduct another group of experiments on trace ali206, which consists of more than 2.8 billion read requests. We record the read count of other blocks in the plane at the last RR (a total of 28507 RRs were triggered). Fig. 5 plots the experimental results, where we only show blocks filled with data. Each small square represents a block, and the color of the small square represents the current disturbance level (i.e., the current read count). Again, in Fig. 5, the value greater than 10 000 is represented by the maximum read count, which is set to 10 000. We can observe that only a few blocks in a plane suffer from severe read disturbance, while most blocks have almost no read disturbance.

These observations reveal that high-popularity data tend to cluster together, resulting in severe read disturbance. Inspired by these compelling findings, we propose a data management strategy of mixing a small amount of hot data proportionally with a large amount of cold data, such that the pressure of read disturbances can be spread over different blocks. Another design principle behind our data management strategy is that erase factor in SSDs should be GC, rather than RR. More

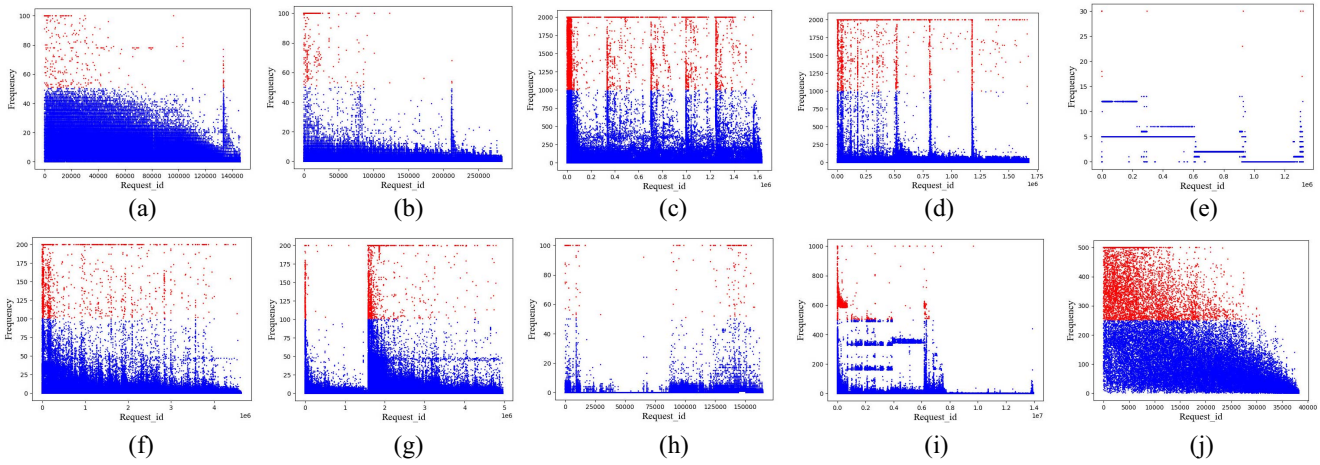


Fig. 4. Data popularity of ten different traces. (a) ads1. (b) ads2. (c) ali206. (d) ali188. (e) msr_web. (f) sys1. (g) sys2. (h) nexus5. (i) usr1. (j) websearch.

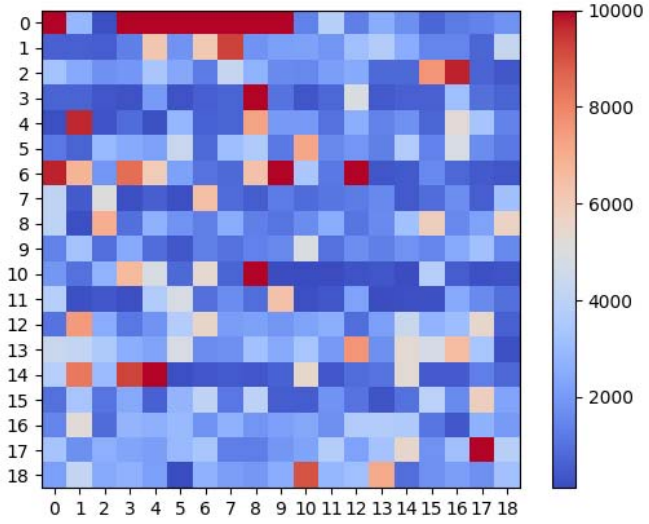


Fig. 5. Heat distribution of read data in a plane.

specifically, it is possible to avoid triggering RR before triggering GC on a block as long as we can configure the read disturbance of the block at low level. It is noteworthy that data in a block can also be recovered from the read disturbance when the block is erased due to GC. See the details of our proposed data management strategy in Section IV.

IV. System Design

A. Write-Back Data Distributing Principles

Now, we introduce the data distributing principle for read-reclaim-induced write-backs in Cocktail. Assuming that each block consists of q pages and is divided into n groups, each group then contains q/n pages; after reading for a period of time, each group causes m page reads; the threshold for triggering an RR operation is $n*m$ reads per block. If a block is heavily read and reclaimed, with k groups of this reclaimed block written into a new block, we can ensure the new block will not trigger a second-time RR as long as it meets the following constraint:

$$k * m + d < n * m \tag{1}$$

where d is the total number of reads for other cold data in the new block. We can derive from Constraint (1) three data distributing principles for read-reclaim-induced write-backs, aiming to reduce the total number of reads for the new block, thereby avoiding the block from triggering a second-time RR.

1) The value of d should be small, that is, there should be enough cold pages in the new block.
2) The value of k should be small, that is, the number of groups allocated to the new block should be small.
3) The value of n should be large. When n becomes larger, the write-backs caused by read-reclaims will be split into more copies and written to more blocks.

It is challenging to place pages in a block to satisfy the above three conditions. Also, it is difficult to predict the popularity of pages (i.e., read-hot pages or read-cold pages). To address these challenges, we propose Cocktail which systematically manages write operations caused by both write requests issued by users (referred to as user-write requests) and write requests triggered by RR (referred to as RR-write requests). Cocktail splits a block into two parts: 1) user-write-part serving user-write requests and 2) RR-part serving RR-write requests. Cocktail writes RR-write requests to the block whose user-write-part has been used up, and such a block is called UW-block. If the portion of user-write-part of each block is relatively large, each block has a smaller number of read operations, thus reducing RR frequency. Three main reasons can explain this trend: 1) Although the data popularity of I/O traces in different environments is different, we can still extract cold-read data from a large number of user-write data [12], [13], [14], [15]; 2) clustering overwritten pages into a block not only reduces the block's read traffic but also increases the number of invalid pages in it; and 3) RR pages are likely to be read-hot pages. To this end, we design a write-pool that maintains a sufficient number of UW-blocks. In addition, we scatter RR-write pages across multiple UW-blocks in the write-pool to further mitigate RR issue.

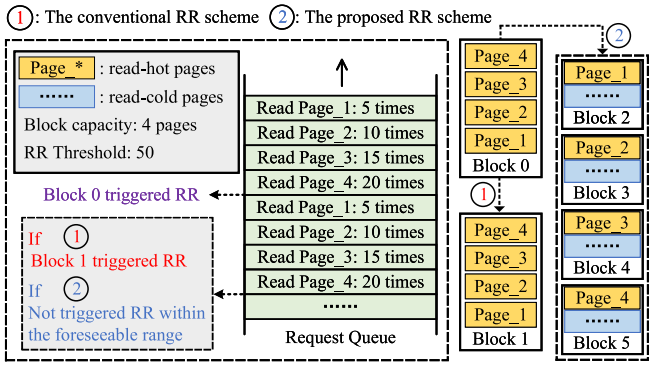


Fig. 6. Compare Cocktail with a conventional RR scheme.

Fig. 6 compares our Cocktail with a conventional read-disturb management scheme. Assuming a block is comprised of four pages, and the RR threshold of a block is set to 50. Given the request queue shown in Fig. 6, after the first four requests are serviced, Block 0 will trigger an RR since the total number of reads for Block 0 (i.e., Page_1–Page_4) is equal to the RR threshold 50. The conventional scheme writes Page_1, Page_2, Page_3, and Page_4 into Block1, which can cause another RR if a set of requests in the same order arrives. However, Cocktail writes these four pages to four different blocks, which reduces the likelihood of the second RR.

The overarching goal of Cocktail is to decentralize hot write-back data of RRs across blocks prefilled with cold data of user requests. There are two challenging issues. First, it is not feasible to predict the popularity of the cold data of user requests. Second, each page requested by a user has its own access pattern, resulting in different popularity [12]. To address these two issues, we equip Cocktail with double write frontiers [15] coupled with a write pool, thereby achieving the goal of dynamically adjusting the ratio of hot write-back data of RRs to cold data of user requests according to the characteristics of RR. In particular, we leverage double write frontiers to separate external user-write requests from internal read-reclaim-induced write-backs. We make use of write frontiers to prefill blocks with relative read-cold data from external user write requests at an adjustable proportion. We keep the prefilled blocks in a corresponding write pool. When the internal hot data of read-reclaim-induced write-backs arrives, we search for the block with the fewest reads in the write pool. In this way, hot write-back data can be easily decentralized and written back to blocks with few reads. As a result, the chance for a block to trigger a second-time RR becomes considerably low. As shown in Section IV-E, the block with the fewest reads in the write pool may contain many invalid pages. Furthermore, even if a second-time RR is triggered, write amplification is still reduced substantially because a small number of hot reads may be put together with many invalid pages in each block.

B. System Overview of Cocktail

The overarching goal of Cocktail is to reduce the number of average reads for blocks by distributing hot writeback data caused by RRs (or RR-write data) across blocks prefilled with cold data of user write requests (or user-write data). We implement Cocktail in the FTL of the tested SSD. Cocktail leverages double write frontiers [12] to prefill blocks, which facilitates creating read-balanced blocks. Cocktail maintains a corresponding write pool to combine user-write data and RR-write data (see Fig. 7). Cocktail utilizes two dynamic thresholds to govern the ratio of user-write data to RR-write data in a block. To balance block-read accesses, Cocktail processes write-backs with respect to two cases, namely, a user-write-intensive case and an RR-write-intensive case.

In the user-write-intensive case, Cocktail writes user-write data to the write frontier block, whereas RR-write data are written to one of the blocks in the write pool. Cocktail keeps writing user-write data into the write frontier block until the amount of the data in the write frontier block reaches the preset threshold. Next, if the write pool is not fully filled, Cocktail will put the write frontier block into the write pool. Otherwise, Cocktail will compare the write frontier block against each block in the write pool to place the blocks with the fewest expected reads in the write pool. If the write frontier block can be placed in the pool, the block exchanged from the pool will become the write frontier. In doing so, Cocktail combines user-write data and RR-write data in the write pool, forging read-balanced blocks. Moreover, Cocktail dynamically adjusts the ratio of user-write data to RR-write data for a block by tuning the two dynamic thresholds. In the RR-write-intensive case, on the other hand, Cocktail performs similar operations except that Cocktail writes RR-write data to the write frontier block while writing user-write data to the blocks in the write pool.

It is worth mentioning that Cocktail will continuously update the blocks in the write pool. There are two purposes for it. The first purpose, as we mentioned before, is to leave relatively cold read data in the write pool. The second purpose is to remove blocks with heavy reads from the write pool to prevent the blocks from triggering RR. As shown in Section II-C, the essence of RR operations and GC operations are erasing blocks. Since blocks in SSD are limited in the number of erasures, each block should be as fully utilized before erasing. However, there still exists free space in the blocks in the write pool. If RR is triggered in these blocks, the free space will be wasted. Therefore, Cocktail will promptly remove the blocks with heavy reads from the write pool and receive user-write data, which ensures that these blocks are fully utilized before being erased.

C. Double Write Frontier

Double write frontier strategy is widely used to reduce write amplification by separating internal and external write requests [12]. We exploit the write frontier to prefill blocks for future data filling in the write pool. We prefill a write frontier block with either user-write data or RR-write data at a given ratio, depending on the characteristics of writeback data. The RR-write frontier is used to receive RR-write data, and the user-write frontier is used to receive user-write data. Since modern SSDs support plane-level parallelism, Cocktail sets up

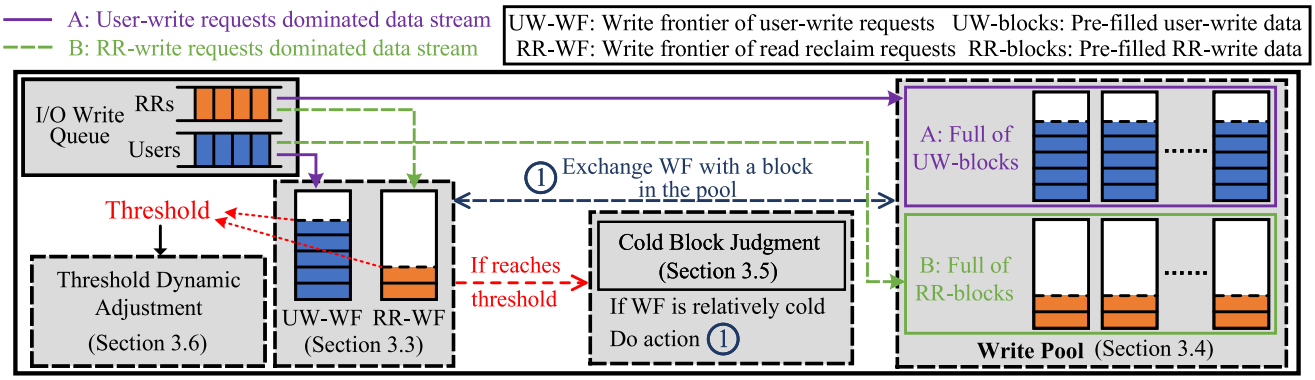


Fig. 7. Cocktail overview.

two write frontiers for each plane, and each plane is associated with an independent write pool. Cocktail also maintains a pool of free blocks. If a write frontier block is prefilled to a predefined threshold, Cocktail will try to exchange the write frontier block with a heavy-read block in the write pool. Or if the write frontier block has been filled, then Cocktail will select one block from the free block pool and then make it a new write frontier block.

To govern the proportions of user-write data and RR-write data in a prefilled block, we introduce two thresholds THuser and THrr. Since a read-balanced block in Cocktail is only comprised of user-write data and RR-write data, we have THuser = (1−THrr) to guarantee the ratio of RR-write data to user-write data in a block remains unchanged regardless of user-write-intensive or RR-write-intensive situations. Recall that in the user-write-intensive case, Cocktail prefills the write frontier block with user-write data. If the amount of data in the block reaches THuser, Cocktail will stop prefilling the write frontier block, put it in the write pool to receive RR-write data for producing a read-balanced block. Cocktail performs similar operations on the write frontier block for RR-write data when working in the RR-write-intensive situation.

D. Write Pool

We set up a write pool where user-write data and RR-write data are written into prefilled blocks, forging read-balanced blocks. We set a flag for each block in the write pool to indicate if a block is prefilled with user-write data or RR-write data. If a block is prefilled with user-write data, Cocktail will fill the block with RR-write data in the write pool, and vice versa.

We employ a sliding window protocol to determine the write pool capacity. The sliding window has a minimum value and a maximum value. We set the minimum value according to realistic experimental situations. Specifically, the minimum value is decided as a floor function of 1% of the number of blocks in a plane. For example, if there are 2048 blocks in a plane, the capacity of the write pool is determined as 20 blocks. Recall from Section III that only a few blocks in a plane suffer from severe read disturb, while most blocks are hardly affected by read disturbance. Moreover, our experiments calculating the number of reads per block when RR is triggered show that the heat distribution is similar to that plotted in Fig. 5. In short, a small ratio can better retain cold data.

When the write pool is full and the user-write frontier block reaches its threshold, Cocktail will increase the write pool capacity by 1 and put the frontier block into the write pool. When the write pool completes an RR-write operation and the number of blocks in the write pool is less than the minimum value, the write pool capacity will be set to the minimum value again to quickly fill the write pool.

Recall from Section IV-B that blocks in the write pool should avoid RR. If the capacity of the write pool is too large, some blocks with many reads may trigger RR while they are still in the write pool. Since determining the maximum value is as difficult as determining a unified overprovisioning in different production environments [16], [17], we set the maximum value to 3% of the plane's block number. The experimental results in Section V show that this configuration is conducive for Cocktail to perform well with ten different traces.

Now, let us demonstrate how Cocktail produces read-balanced blocks with respect to the following two cases.

Case 1 (User-Write Data Processing): The primary goal of processing user-write data is to combine such data with prefilled RR-write data into read-balanced blocks. To this end, we propose Algorithm 1 that facilitates user-write data processing.

When a request of user-write data arrives, Cocktail first searches the write pool for an available block prefilled with RR-write data (see line 3 in Algorithm 1). If such a block is found, Cocktail populates the user-write data into the found block, thereby generating a read-balanced block by combining the newly written user-write data with the prefilled RR-write data (see line 37 in Algorithm 1). Otherwise, with no such block found in the write pool, Cocktail writes the user-write data into the write frontier block until it reaches the predefined threshold THuser (see lines 13 and 14 in Algorithm 1). When THuser is reached, Cocktail checks the write pool to determine whether the number of blocks in the pool has reached its capacity. If the pool is not full, Cocktail will put this write frontier block directly into the pool, then apply for a free block from free block pool and makes it a new write frontier block. If the pool is full, Cocktail will compare the write frontier block with the existing blocks in the pool in terms of future block reads. As a result, blocks with fewer reads will stay in the pool, whereas the block with the highest future reads will serve as

Algorithm 1 User-Write Data Write Strategy

```
1: /*RR_blocks are pre-filled RR-write data*/
2: /*WR_blocks are pre-filled user-write data*/
3: if RR_block can be found in pool then
4:     target_block = RR_block
5: else
6:     target_block = nullptr
7: end if
8: /*wf is the write frontier of user-write data*/
9: /*bk_pool is the blocks number of pool*/
10: /*num_blocks is the blocks number of plane*/
11: /*num_pages is the pages number of block*/
12: if target_block == nullptr then
13:     write user-write data to wf
14:     wf.index++
15:     if bk_pool < max_num_blocks then
16:         if wf.index >= TH_user * num_pages then
17:             /*It will be a WR_block*/
18:             wf.status = State_WR
19:             pool.push(wf)
20:             wf = get_a_new_block()
21:             check GC required
22:         end if
23:     else
24:         if wf.index == num_pages then
25:             wf = get_a_new_block()
26:             check GC required
27:         else if wf.index >= TH_user * num_pages then
28:             Find(target_block)
29:             if target_block != nullptr then
30:                 pool.remove(target_block)
31:                 pool.push(wf)
32:                 wf = target_block
33:             end if
34:         end if
35:     end if
36: else
37:     write user-write data to target_block
38:     target_block.index++
39: end if
```

Algorithm 2 RR-Write Data Write Strategy

```
1: /*RR_blocks are pre-filled RR-write data*/
2: /*WR_blocks are pre-filled user-write data*/
3: if WR_block can be found in pool then
4:     target_block = WR_block
5: else
6:     target_block = nullptr
7: end if
8: /*wf is the write frontier of RR-write data*/
9: /*bk_pool is the blocks number of pool*/
10: /*num_blocks is the blocks number of plane*/
11: /*num_pages is the pages number of block*/
12: if target_block == nullptr then
13:     if bk_pool < max_num_blocks then
14:         write RR-write data to wf
15:         wf.index++
16:         if wf.index == num_pages then
17:             wf = get_a_new_block()
18:             check GC required
19:         else if wf.index >= TH_rr * num_pages then
20:             /*It will be a RR_block*/
21:             wf.status = state_RR
22:             pool.push(wf)
23:             wf = get_a_new_block()
24:             check GC required
25:         end if
26:     else
27:         Find(target_block)
28:         write RR-write data to target_block
29:         target_block.index++
30:         if target_block.index == num_pages then
31:             pool.remove(target_block)
32:         end if
33:     end if
34: else
35:     write RR-write data to target_block
36:     target_block.index++
37: end if
```

the write frontier block (see lines 15–35 in Algorithm 1). It is noteworthy that there exist two factors that significant affect the number of future reads for a block. One is the number of current reads for a block, and the other is the number of invalid pages in a block.

Case 2 (RR-Write Data Processing): Cocktail follows a similar procedure to generate read-balanced blocks when processing RR-write data. Algorithm 2 presents the pseudo-code of Cocktail's RR-write data procedure. More specifically, when a request of RR-write data arrives, Cocktail will first search for a block prefilled with user-write data in the pool (see line 3 in Algorithm 2). Then, if such a block is found, Cocktail writes the arriving data into the found block; if not, Cocktail writes the arriving data in the write frontier block. Next, when the write frontier block reaches the RR-write data threshold THrr, Cocktail will try to place it in the dynamic pool it the pool is not full (see lines 35 and 13–25 in Algorithm 2). If the pool is full of blocks prefilled with RR-write data, RR dominates the data stream, causing a large number of data writes. In order to avoid polluting too many blocks, Cocktail will aggregate data into several blocks. Specifically, Cocktail picks from the pool the block with the least number of reads, and then fills it with the arriving RR-write data (see lines 27–32 in Algorithm 2).

E. Read Reclaim Count Factors

The principle of block replacement is to keep the blocks with fewer future reads in the write pool. Though it is not feasible to know the number of future reads for a block, we observe that two key factors have significant impacts on RR count. The two key factors are the total number of current reads for a block and the number of valid pages in the block.

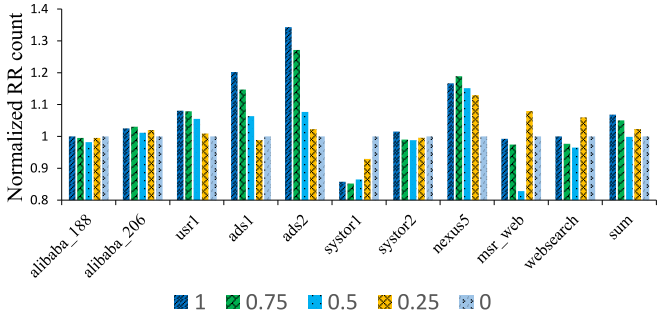


Fig. 8. Triggered RR count with respect to Ratio$_{(BR/IP)}$.

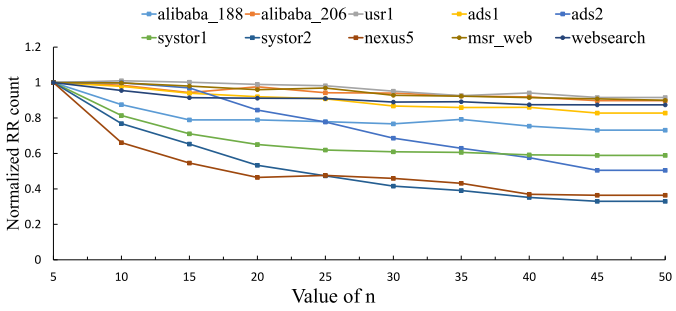


Fig. 9. Triggered RR count with respect to n.

To facilitate predicting the number of RRs (or RR count), we introduce an RR count factor, which expresses the joint impacts of the above-mentioned two key factors

$$\text{factor} = \text{ratio} * (RC/\text{TH}) + (1 - \text{ratio}) * (Pv/Pa) \quad (2)$$

where TH denotes the RR threshold; RC denotes the total number of block reads; and Pv and Pa represent the number of valid pages and the number of all pages in a block, respectively. By calculating RC/TH and Pv/Pa, the two different coefficients of read count and valid pages can be normalized. The parameter ratio is an additional variable. By setting the ratio, the factor of each block can be adjusted. This factor determines which block will be selected in the Cocktail strategy.

To find out the best *factor* configuration to reduce RR count, we conduct a group of experiments under ten real-world workload conditions. We investigate the impact of *factor* on RR count by varying *ratio* from 0 to 1 with an increment of 0.25.

Fig. 8 reveals that among the five configurations (i.e., 1, 0.75, 0.5, 0.25, and 0), ratio 0.5 is the best configuration for *factor*. This finding is reasonable. Fig. 8 shows that ratio 0.5 leads to the least RR count under most workload conditions. Moreover, because both the number of block reads and the number of invalid pages has significant impacts on reducing RR count, it is judicious to consider these two factors equally. As we mentioned before, different IO-traces have different data distributions. Therefore, in a real production environment, this parameter can be adjusted according to data characteristics. In this study, we will set the ratio of the total number of reads to the number of valid pages for a block to 0.5.

F. Proportions of User-Write Data and RR-Write Data

The proportions of user-write data and RR-write data in a block play a role in reducing RR count and creating read-balanced blocks. To gain the insights into the impact of the data proportion, we carry out a set of experiments to obtain the best data proportion for generating read-balanced blocks. As mentioned in Section IV-A, assuming that a block is divided into n parts, and each block consists of 1/n RR-write data and $(n-1)/n$ user-writer data. We examine RR count with respect to different n values under ten workload conditions (see Fig. 9).

Fig. 9 indicates that as the proportion of RR-write data decreases, RR count goes down as well, eventually showing a flat trend. When choosing the best configuration n, not only the findings from Fig. 9 but also other factors are worthy of considering. In particular, if very few blocks exist in the write pool, we can derive that the current data stream has few user-write data. A small n value helps Cocktail place the write frontier block early into the write pool to combine with RR-write data, creating read-balanced blocks. On the contrary, if the pool is full, we can infer that user-write data is far more than RR-write data in the current data stream. Accordingly, we can use a relatively high n to select "high-quality" blocks to combine RR-write data.

Rather than using a fixed n, we dynamically adjust the proportions of user-write data and RR-write data in a block. Assuming that the write pool can store m blocks at most, when an RR triggers and the pages in the block where the RR triggers are all valid, in order to make sure all RR-write data have a place to be written in the write pool, each block in the write pool needs to reserve $1/m$ of space, i.e., $n = m$. As mentioned in Section IV-D, the capacity of the write pool changes dynamically. When $n = m$, the value of n can dynamically change with the capacity of the write pool. For example, if there are 20 blocks in the write pool, the proportion of user-write data in each block is 19/20, while the proportion of RR-write data is 1/20. That is, the proportion of RR-write data is the reciprocal of write pool capacity. The final experimental results in Section V show that Cocktail performs well with ten different traces.

G. Analysis

Our trace observations (800 traces in Alibaba center in 2020 [18]) reveal that real-world traces are comprised of both read and write requests. Traces in many other scenarios [19], [20], [21], [22] also contain a large proportion of write requests. In these scenarios, Cocktail can adequately preserve relatively cold data from written data. These preserved cold data can be combined with the hot read data produced by RR. Though Cocktail may not work in the read-only request scenario (a rare scenario), this can be solved by other algorithms chosen via a selection mechanism in FTL. In short, Cocktail focuses on scenarios where read and write requests coexist.

We analyze the overhead of Cocktail in terms of computation cost and memory requirement. The main computation cost in Cocktail is to find the most suitable block in the pool when a request of user-write data arrives. Since the size of the pool is

TABLE I
CONFIGURATION OF THE SIMULATED SSD

SSD organization	8 channels 4 chips per channel 2 dies per chip
Flash Microarchitecture	2 planes per die 2048 blocks per plane 256 pages per block Page size: 8 KB
Flash Latencies	Read latency: 75 us ECC latency: 20 us Program latency: 750 us Erase latency: 3.8 ms
GC Parameters	Policy: GREEDY Threshold: 0.28

TABLE II
CHARACTERISTICS AND PROCESSING METHODS OF WORKLOADS

Trace	Read Ratio	RAF	Repeat Times
$alibaba\_206$	0.8648	2	1
$alibaba\_188$	0.6539	4	1
$usr1$	0.9146	8	1
$systor1$	0.6217	30	5
$systor2$	0.5996	30	5
$ads1$	0.9052	40	10
$ads2$	0.5625	40	10
$nexus5$	0.3376	50	10
msr_web	0.9925	50	10
$websearch$	0.9997	50	10

set to be relatively small (such as 2% of the number of blocks in a plane), the search can be completed very quickly. Note that regardless of the FTL design, a counter update/check must be performed for each read request. The above operations are typically performed in nanoseconds, while NAND-related operations are typically performed in microseconds [23], [24], [25]. As a result, the proposed design only incurs negligible computation overhead since those essential NAND-related operations (i.e., moving unreliable pages due to read disturbance) require much longer time.

Cocktail does not consume extra SSD storage space. The write pool in Cocktail is a simple implementation of an array of pointers. Because of this, Cocktail can even reduce the length of the pointer array to 0 when the SSD storage space is not enough to store data. For the memory requirement, the write pool in Cocktail exists logically (simply pointing to the actual block through the pointer), which means that the design of the pool does not take up additional memory overhead.

V. EXPERIMENTAL RESULTS

We implement Cocktail within simulator MQSim [26]. MQSim is a well-known open source SSD simulator that accurately simulates modern 3-D NAND flash memory. We set the RR threshold per block to 40*$\text{Num}_{\text{pages}}$ [3]. Table I summarizes the configuration details of our simulated SSD.

Workloads: We conduct experiments on ten real-world workload traces of various read/write ratios (see Table II). Among these ten traces, *alibaba_188* and *alibaba_206* are one-month access behaviors of two different virtual machines in Alibaba center in 2020 [18]. *systor1* and *systor2* record the data access behaviors of an enterprise virtual desktop infrastructure in two different months of 2017 [19]. *nexus5* is a trace of data accesses on smart phones [20]. *ads1* and *ads2* are from Microsoft's Ads Platform [21]. *usr1* and *msr_web* are from an enterprise cluster [22]. *websearch* is from the trace repository of UMass.

Since RR is only triggered in read-intensive tasks, workload traces collected in small devices or short collection times may not trigger sufficient RR operations. To address this issue, we increase the execution time on such traces by a factor of *N* times, where *N* is the repeat times shown in Table II. We also increase the density of read requests per trace according to the data popularity of the ten traces shown in Fig. 4. In addition, we utilize the number of repeated times for a read request—*RAF*—to ensure each trace has a relatively similar frequency of reading requests. We configure the number of times an amplified trace is run (*Repeated Times*) to set a relatively similar amount of data for the ten traces.

Comparison of Methods: We evaluate the performance of Cocktail by comparing it with three different solutions. We set the double write frontier method as a baseline solution. Besides, we evaluate our Cocktail against another two disturbance management solutions redFTL and IPR [7], [9].

The double write frontier method makes use of write frontiers to write user-write data and RR-write data to different blocks. redFTL records the number of page reads by adding one byte to the LPA of each page. Based on this extra information, redFTL combines RR-write data with the block with least read count to create read-balanced blocks when performing RR. redFTL increases the RR threshold of read-balanced blocks to twice the normal RR threshold by changing the read voltage of read-balanced blocks, aiming to avoid read-balanced blocks from second-time RR. When RR triggers again in read-balanced blocks, the RR-write data will be written to blocks whose RR threshold is five times the normal RR threshold.

IPR strives to create read-balanced blocks by monitor-blocks and IPR-blocks. Monitor-blocks keep monitoring the read count of each page. When performing RR, if the read count of a page is higher than the average read count, the page will be rewritten to an IPR-block; otherwise, the page will be written back to an ordinary block. Similar to redFTL, IPR also increases the RR threshold of IPR-blocks to ten times the normal RR threshold by changing the read voltage of these blocks. Monitor-blocks account for 10% of the capacity, and IPR-blocks account for 15% of the capacity.

Our Cocktail is distinctly different from redFTL and IPR in that Cocktail preserves a small portion of each block for future combination with hot RR-write data to create read-balanced blocks, thereby distributing hot RR-write data across a wide range of blocks. Unlike redFTL and IPR that may centralize all hot data in a small number of blocks, Cocktail is conducive to reducing the number of RRs. Such a reduction cuts back not only SSD response time but also the number of erase operations. It is well known that the time required for a read request mainly lies in the read speed of the storage medium. Reducing the number of block-erases can minimize the impact of write amplification on user requests. Besides, alleviating

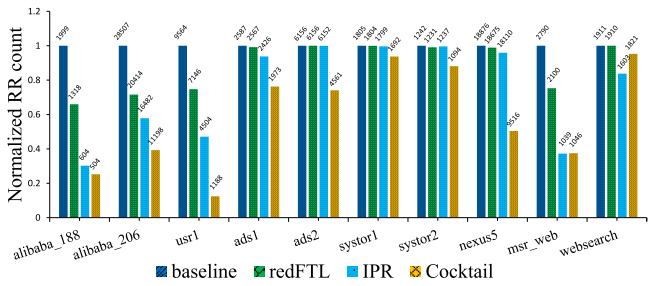


Fig. 10. Normalized RR count.

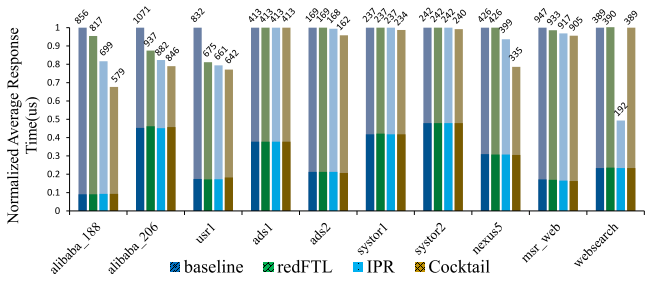


Fig. 11. Normalized average response time.

read disturbance can also reduce the number of ECC retry times, which further improves I/O efficiency.

The experimental results can be found in the following three sections. In particular, we evaluate Cocktail in terms of three aspects: 1) RR count; 2) response time of SSD; and 3) write amplification and GC count.

A. RR Count

In this group of experiments, we focus on the impact of Cocktail on service life of SSD. We compare Cocktail with the baseline solution (i.e., double write frontier), redFTL, and IPR under ten workload conditions in terms of RR count. Because the experimental results of the four algorithms scatter on different scales. We normalize RR count results to a common scale by setting the baseline solution's results to 1 and proportionally scale the results of redFTL, IPR, and Cocktail.

Fig. 10 plots the normalized RR counts of the four algorithms under ten workload conditions. Compared to the other three algorithms, Cocktail performs better under many workload conditions. Under workload condition *usr1*, Cocktail reduces the RR count of the other three solutions by more than 80%. The superiority of Cocktail is expected. The centralized placement strategy for RR-write data used by existing solutions such as redFTL and IPR may trigger a second RR. Unlike these solutions, Cocktail judiciously distributes RR-write data across a broad range of blocks by combing RR-write data with user-write data, thereby remarkably avoiding occurrences of the second RRs. Moreover, the decentralized RR-write-data placement strategy in Cocktail also reduces the overall block reads. But in the scenario of only read requests, Cocktail does not have enough user-write data to combine with RR-write data. While Cocktail has a mechanism for dealing with this scenario (case 2 in Section IV-D), although the effect is not as dramatic as it normally does. Cocktail outperforms the baseline and redFTL except IPR when dealing with the *msr_web* trace and *websearch* trace.

In short, Cocktail reduces not only the chance of the second RR caused by the centralized placement but also the overall block reads. These salient features enable Cocktail to reduce the average RR count of the baseline solution, redFTL, and IPR by 40.77%, 36.06%, and 21.75%, respectively.

B. Response Time

We pay attention to the performance of Cocktail in terms of SSD response time. We examine the normalized response time and 99-percentile to 100-percentile tail latency of Cocktail, the baseline solution, redFTL, and IPR under the ten workload conditions. A read frequently incurs multiple retry steps. Under a 3-month data retention period for zero P/E cycles (i.e., at the beginning of SSD lifetime), every read requires more than three retry steps [1]. Based on this finding, we retry ECC once when a block's read count reaches 70% of the RR threshold, two times when it reaches 80%, and three times when it reaches 90%. Fig. 11 depicts the experimental results. The darker part of the histograms represents the average ECC-retry time per request. It shows that in cases *alibaba_206*, *systor1*, and *systor2*, the average ECC-retry time accounts for nearly half of the average response time.

Figs. 11 and 12 show the average response time and 99-percentile to 100-percentile tail latency of the four methods on the ten workloads. It indicates that Cocktail is superior to the baseline solution, redFTL, and IPR in terms of average response time and tail latency under *ali206*, *ali188*, *msr_web* and *nexus5*. When testing on traces *ads1*, *ads2*, *systor1*, and *systor2*, Cocktail has no obvious advantage over the other three methods. These observations are reasonable. This is because SSDs have to transfer data from reclaimed block to other blocks during RR. SSD's endeavor to ensure the atomicity of operations causes an inevitably delay on subsequent read requests. The delay becomes more serious when working under read-intensive conditions. Unlike Cocktail, the traditional solutions fail to reduce RR count, which leads to the longer response times under other workload conditions except *websearch*. Under *websearch*, IPR's performance is slightly better than Cocktail.

C. Write Amplification and GC Count

Now, we are at the position of evaluating the performance of Cocktail in terms of write amplification and GC count. Again, we compare Cocktail with the other three solutions under the ten workload conditions. The experiment results plotted in Figs. 13 and 14 unveil that although Cocktail mainly focuses on read-disturb management rather than GC optimization, yet Cocktail reduces write amplification of erase operations and the number of GCs. Figs. 13 and 14 also list the block-erase counts of the four schemes with respect to the ten different traces.

An important goal of GC algorithms is to reduce write amplification. For this reason, a number of GC algorithms have been proposed to archive this goal. For example, the

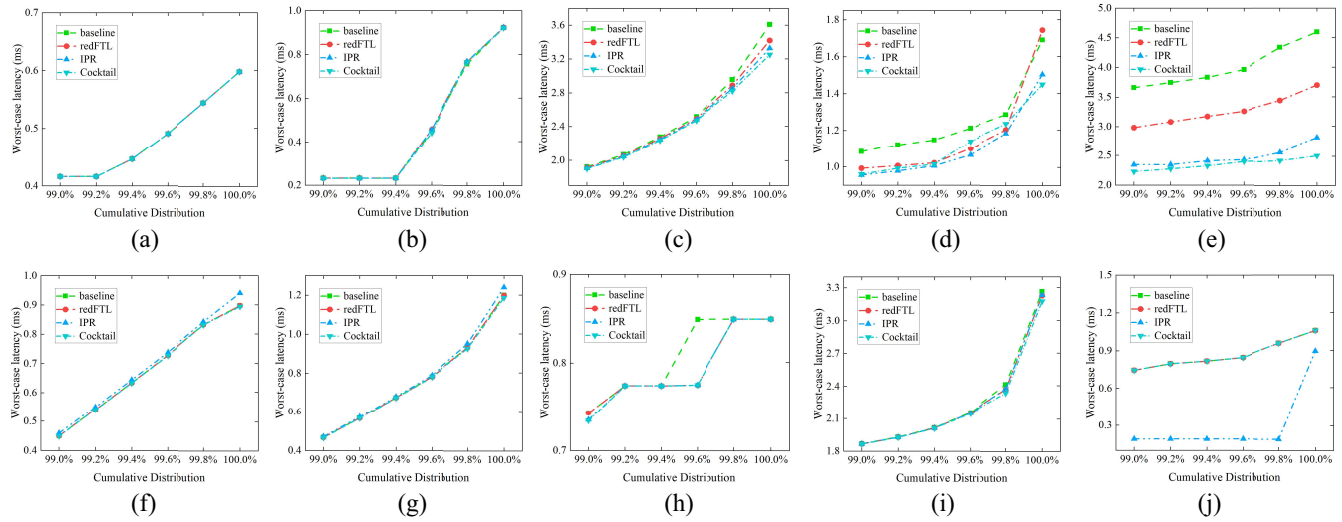


Fig. 12. Worst case latency of ten different traces. (a) ads1. (b) ads2. (c) ali206. (d) ali188. (e) msr_web. (f) sys1. (g) sys2. (h) nexus5. (i) usr1. (j) websearch.

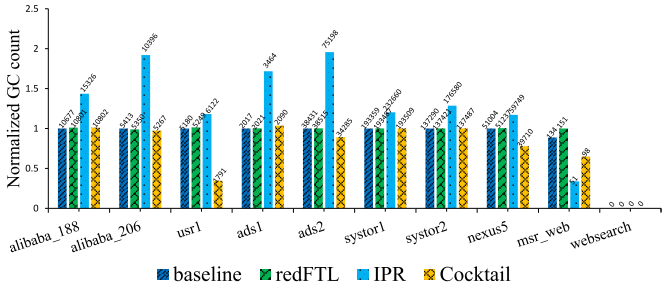


Fig. 13. Normalized GC count.

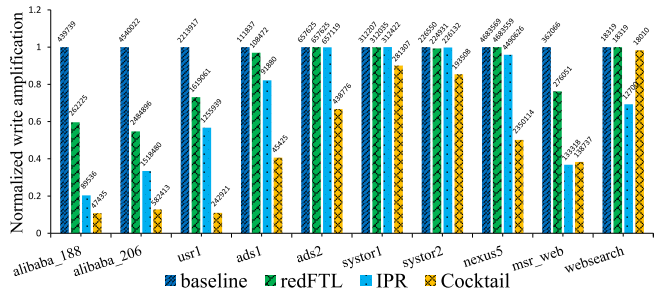


Fig. 14. Normalized write amplification.

FIFO GC algorithm [17], which selects blocks in a cyclic manner, is easy to implement but delivers the worst write amplification. Greedy GC algorithm [17] that selects the block containing the least number of valid pages is designed to minimize write amplification in uniform random write scenarios. Windowed GC algorithm [16] is devised to combine the simplicity of the FIFO algorithm with Greedy algorithm's the superior performance. Van Houdt [27] developed d-choices GC algorithm, aiming to select the block with the least number of valid pages from a set of d randomly chosen blocks. Since even for moderate d values (e.g., d = 10), d-choices GC algorithm performs slightly worse than the Greedy algorithm, we adopt the Greedy algorithm in this study to effectively reduce write amplification.

Fig. 13 depicts the GC count of the four compared solutions. It can be seen from Fig. 13 that Cocktail surpasses the other three solutions and generates fewer GCs in different workloads. For instance, the number of GCs triggered by Cocktail is less than that of the baseline, redFTL, and IPR by an average of 12.29%, 16.21%, and 22.40%, respectively. When testing in workload *usr1* IPR delivers the worst performance, causing the highest GC count. The main cause is that IPR increases its read threshold at the expense of partial block space and only used to store hot read data. As a result, when writing the same amount of data, IPR triggers more GCs because of its small space.

Fig. 14 shows the write amplification of the four solutions. It demonstrates that Cocktail outmatches the other three solutions: the average number of transferred pages during erasure in Cocktail is less than that in other methods. Specify, Cocktail reduces the write amplification by 49.57%, 45.99%, and 29.93% compared with the baseline, redFTL, and IPR, respectively.

In a nutshell, Cocktail effectively reduces write amplification by slashing the number of RRs. In particular, most of the data in a block that causes RR is hot data. These hot data remain valid in their blocks until their blocks are reclaimed, which causes more data to be transferred when performing block erase operations. In other words, the write amplification caused by RR is more severe than that caused by GC. Therefore, our Cocktail advocates that reducing the number of RRs can substantially alleviate write amplification.

VI. Related Work

A lot of attention has been paid to solving the read-disturb issue in SSDs. For example, Zambelli et al. [28] illustrated different behaviors of TLC NAND flash-based SSDs under uniform and concentrated read-disturb conditions. These findings demonstrate the impacts of workload models on the reliability of enterprise SSDs. Li et al. [29] built a model to evaluate the read-disturb level of blocks, which facilities migrating frequently read data to read-tolerant blocks.

In addition, Li et al. [29] and Kang et al. [30] introduced reinforcement learning into RR and GC scheduling, aiming to perform data move and erase operations in the time intervals between I/O requests. Zhao et al. [31] proposed a data reallocation scheme that distributes the data of RR blocks across different blocks, thereby reducing read refresh operations and raw bit errors. The rationale behind Zhao's scheme is to make each block reach its own optimal read count by considering PE cycles and block read counts. In short, prior studies strive to solve the read-disturb problem mainly from two aspects: 1) firmware-level correction and 2) read-disturb management.

Firmware-Level Correction: Many researchers attempt to untangle the problem through internal correction methods at the firmware level. Researchers have conducted a series of firmware-level experiments, showing that the read-disturb effect has a significant adverse impact on the SSD erase cycle, data retention time, read response time, etc. [32], [33]. To this end, Kang et al. [32] proposed a firmware-level technique that adds an extra virtual unit to each unit NAND string, aiming to reduce the read disturbance during read operations. Though this technique can mitigate the read-disturb problem to some extent, the costly NAND flash modifications constrain its applicability to a narrow range. Some other studies tried to mitigate the read-disturb problem with a strategy of reducing the pass-through voltage [7], [33]. However, this pass-through-voltage-reducing is premised on a superior error correction capability of ECC and obtains read-disturb alleviation at the expense of increasing page errors. Specifically, reading pages with the normal pass-through voltage leads to few page errors. As the pass-through voltage reduces, the read-disturb problem is alleviated, however, the number of page errors increases considerably. In addition to this increasing number of page errors, the pass-through-voltage-reducing strategy also relies heavily on a powerful ECC engine, which may not always be available. Moreover, the pass-through-voltage-reducing strategy is conflicted with other optimization objectives, such as improving SSD reliability and reducing SSD read latency [1]. In short, it is difficult to adopt the pass-through-voltage-reducing strategy merely to alleviate the read-disturb problem in the actual environment.

Device-Level Management: Since firmware-level correction methods come with some negative impacts on other aspects of SSDs, recent researchers embarked on read-disturb management techniques to reduce the number of triggered RRs. For example, Liu et al. [8] proposed a decentralized read-disturb management method that distributes write-back data of RRs to different blocks, striving to prevent hot read pages of a block from being written back to the same block. Unfortunately, Liu falls short on considering hot read pages from other blocks, so the likelihood of aggregation of hot read pages is still very high. Ha et al. [7], [34] developed a read-disturb management technique called *redFTL*, which increases the maximum readable count of some blocks, and then writes the RR data back into these blocks. Wu et al. [9] built a read-disturb management approach named *IPR* that increases the maximum readable count of blocks at the cost of some block capacity. Similar to redFTL, IPR also writes all RR data back into a small number of changed blocks. Although these two similar methods have relatively good results in some application scenarios, they are not very adaptable and versatile and perform poorly in many scenarios. The main cause for their poor performance is that making the hot-read-page prediction based on the number of reads of read-reclaimed pages has low accuracy. Therefore, hot read pages in these two methods are still centralized in a small number of blocks, triggering second-time RRs and giving birth to a high number of page reads.

VII. Conclusion

In this article, we proposed an efficient and low-cost read-disturb management technique called *Cocktail*. Cocktail reduces the number of RRs, aiming to alleviate read-disturb effects. In particular, Cocktail extracts cold data of user requests and combines it with the write-back data caused by RR. Cocktail distributes hot data in a wide range of blocks, avoiding the centralization of hot data in a small number of blocks.

To quantitatively evaluate the performance of Cocktail, we conducted extensive experiments under ten real-world workload conditions. We compared Cocktail with three other schemes baseline, redFTL, and IPR in terms of SSD service life, SSD response time, write amplification, and the number of GCs. The experimental results showed that Cocktail outperforms the three compared schemes. In particular, compared with the three alternative schemes, Cocktail reduces the number of RRs, the average response time, the 99-percentile tail latency, and the number of GCs by an average of 40.77%, 10.82%, 5.40%, and 12.29%, respectively. Cocktail also alleviates the write amplification of the three alternative schemes by an average of 49.57%.

References

[1] J. Park, M. Kim, M. Chun, L. Orosa, J. Kim, and O. Mutlu, "Reducing solid-state drive read latency by optimizing read-retry," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 702–716.

[2] W. Liu et al., "DEPS: Exploiting a dynamic error prechecking scheme to improve the read performance of SSD," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 40, no. 1, pp. 66–77, Jan. 2021.

[3] C.-Y. Liu, Y. Lee, M. Jung, M. T. Kandemir, and W. Choi, "Prolonging 3D NAND SSD lifetime via read latency relaxation," in *Proc. 26th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2021, pp. 730–742.

[4] S. Wu et al., "GC-steering: GC-aware request steering and parallel reconstruction optimizations for SSD-based RAIDs," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 4587–4600, Dec. 2020.

[5] Y. Lv, L. Shi, L. Luo, C. Li, C. J. Xue, and E. H.-M. Sha, "Tail latency optimization for LDPC-based high-density and low-cost flash memory devices," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 544–557, Mar. 2022.

[6] F. Yang, Y. Chen, H. Mao, Y. Lu, and J. Shu, "ShieldNVM: An efficient and fast recoverable system for secure non-volatile memory," *ACM Trans. Storage*, vol. 16, no. 2, pp. 1–31, 2020.

[7] K. Ha, J. Jeong, and J. Kim, "An integrated approach for managing read disturbs in high-density NAND flash memory," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 35, no. 7, pp. 1079–1091, Jul. 2016.

[8] C.-Y. Liu, Y.-M. Chang, and Y.-H. Chang, "Read leveling for flash storage systems," in *Proc. 8th ACM Int. Syst. Storage Conf.*, 2015, pp. 1–10.

[9] T.-C. Wu, Y.-P. Ma, and L.-P. Chang, "Flash read disturb management using adaptive cell bit-density with in-place reprogramming," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 325–330.

[10] W. Liu et al., "Characterizing the reliability and threshold voltage shifting of 3D charge trap NAND flash," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 312–315.

[11] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai, "Threshold voltage distribution in MLC NAND flash memory: Characterization, analysis, and modeling," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2013, pp. 1285–1290.

[12] B. Van Houdt, "Performance of garbage collection algorithms for flash-based solid state drives with hot/cold data," *Perform. Eval.*, vol. 70, no. 10, pp. 692–703, 2013.

[13] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proc. 8th IEEE Real-Time Embedded Technol. Appl. Symp.*, 2002, pp. 187–196.

[14] M.-L. Chiang, P. C. Lee, and R.-C. Chang, "Using data clustering to improve cleaning performance for flash memory," *Softw. Pract. Exp.*, vol. 29, no. 3, pp. 267–290, 1999.

[15] B. Van Houdt, "On the necessity of hot and cold data identification to reduce the write amplification in flash-based SSDs," *Perform. Eval.*, vol. 82, pp. 1–14, Dec. 2014.

[16] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Write amplification analysis in flash-based solid state drives," in *Proc. SYSTOR Israeli Exp. Syst. Conf.*, 2009, pp. 1–9.

[17] P. Desnoyers, "Analytic modeling of SSD write performance," in *Proc. 5th Annu. Int. Syst. Storage Conf.*, 2012, pp. 1–10.

[18] J. Li, Q. Wang, P. P. Lee, and C. Shi, "An in-depth analysis of cloud block storage workloads in large-scale production," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, 2020, pp. 37–47.

[19] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 1–11.

[20] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/O characteristics of smartphone applications and their implications for eMMC design," in *Proc. IEEE Int. Symp. Workload Characterization*, 2015, pp. 12–21.

[21] S. Kavalanekar, B. Worthington, Q. Zhang, and V. Sharda, "Characterization of storage workload traces from production windows servers," in *Proc. IEEE Int. Symp. Workload Characterization*, 2008, pp. 119–128.

[22] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, 2008.

[23] N. Shibata et al., "13.1 A 1.33-Tb 4-bit/cell 3-D flash memory on a 96-word-line-layer technology," *IEEE J. Solid-State Circuits*, vol. 55, no. 1, pp. 178–188, Jan. 2020.

[24] C. Kim et al., "A 512-Gb 3-b/cell 64-stacked WL 3-D-NAND flash memory," *IEEE J. Solid-State Circuits*, vol. 53, no. 1, pp. 124–133, Jan. 2018.

[25] S. Lee et al., "A 1Tb 4b/cell 64-stacked-WL 3D NAND flash memory with 12MB/s program throughput," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2018, pp. 340–342.

[26] A. Tavakkol, J. Gómez-Luna, M. Sadrosadati, S. Ghose, and O. Mutlu, "MQSim: A framework for enabling realistic studies of modern multi-queue SSD devices," in *Proc. 16th USENIX Conf. File Storage Technol.*, 2018, pp. 49–66.

[27] B. Van Houdt, "A mean field model for a class of garbage collection algorithms in flash-based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 191–202, 2013.

[28] C. Zambelli, P. Olivo, L. Crippa, A. Marelli, and R. Micheloni, "Uniform and concentrated read disturb effects in mid-1X TLC NAND flash memories for enterprise solid state drives," in *Proc. IEEE Int. Rel. Phys. Symp. (IRPS)*, 2017, p. 5.

[29] J. Li et al., "Mitigating negative impacts of read disturb in SSDs," *ACM Trans. Design Autom. Electron. Syst.*, vol. 26, no. 1, pp. 1–24, 2020.

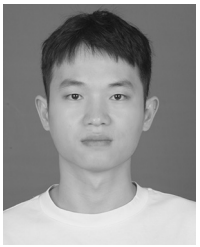
[30] W. Kang, D. Shin, and S. Yoo, "Reinforcement learning-assisted garbage collection to mitigate long-tail latency in SSD," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5, pp. 1–20, 2017.

[31] M. Zhao, J. Li, Z. Cai, J. Liao, and Y. Shi, "Block attribute-aware data reallocation to alleviate read disturb in SSDs," in *Proc. Design, Autom. Test Europe Conf. Exhibit. (DATE)*, 2021, pp. 1096–1099.

[32] M. Kang et al., "Improving read disturb characteristics by self-boosting read scheme for multilevel NAND flash memories," *Jpn. J. Appl. Phys.*, vol. 48, no. 4S, 2009, Art. no. 4C062.

[33] Y. Cai, Y. Luo, S. Ghose, and O. Mutlu, "Read disturb errors in MLC NAND flash memory: Characterization, mitigation, and recovery," in *Proc. 45th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2015, pp. 438–449.

[34] K. Ha, J. Jeong, and J. Kim, "A read-disturb management technique for high-density NAND flash memory," in *Proc. 4th Asia-Pacific Workshop Syst.*, 2013, pp. 1–6.



Genxiong Zhang received the B.E. degree in computer science and technology from South China Agricultural University, Guangzhou, China.

He is a Research Student with the Computer Science Department, Jinan University, Guangzhou, China. His current research interests include storage system and nonvolatile memory.

Yuhui Deng received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, Wuhan, China, in 2004.

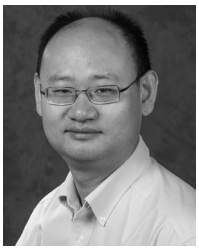
He is a Professor with the Computer Science Department, Jinan University, Guangzhou, China. Before joining Jinan University, he worked as a Senior Research Scientist with EMC Corporation, Beijing, China, from 2008 to 2009. He worked as a Research Officer with Cranfield University, Bedford, U.K., from 2005 to 2008. His research interests cover information storage, cloud computing, green computing, computer architecture, and performance evaluation.

Yi Zhou received the B.E.E. and M.S.E.E. degrees in electronic engineering from Beijing University of Technology, Beijing, China, in 2006 and 2010, respectively, and the Ph.D. degree in computer science from Auburn University, Auburn, AL, USA, in 2018.

He is currently an Assistant Professor with the TSYS School of Computer Science, Columbus State University, Columbus, GA, USA. His research interests include energy-saving techniques, database systems, big data techniques, and parallel computing.

Shujie Pang received the M.S. degree from the Computer Science Department, Yantai University, Yantai, China. He is currently pursuing the Ph.D. degree from the Computer Science Department, Jinan University, Guangzhou, China.

His current research interests include storage system, computer architecture, and performance evaluation.



Jianhui Yue received the Ph.D. degree in computer science from The University of Maine, Orono, ME, USA, in 2012.

He is an Assistant Professor with the Computer Science Department, Michigan Technological University, Houghton, MI, USA. Before joining Michigan Technological University, he was a Visiting Assistant Professor with Miami University, Oxford, OH, USA. His research interests include computer architecture and systems.

Dr. Yue received the Best Paper Award at IEEE CLUSTER 2007 and was the Best Paper Award Candidate at HPCA 2013.

Yifeng Zhu received the B.S. degree in electrical engineering from the Huazhong University of Science and Technology, Wuhan, China, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Nebraska, Lincoln, NE, USA, in 2002 and 2005, respectively.

He is Professor with the Department of Electrical and Computer Engineering, The University of Maine, Orono, ME, USA. His current research interests include computer architecture and systems, data storage systems, and energy-efficient memory systems.