Reparo: A Fast RAID Recovery Scheme for Ultra-large SSDs

DUWON HONG, Seoul National University, Korea
KEONSOO HA and MINSEOK KO, Samsung Electronics, Korea
MYOUNGJUN CHUN and YOONA KIM, Seoul National University, Korea
SUNGJIN LEE, Daegu Gyeongbuk Institute of Science and Technology (DGIST), Korea
JIHONG KIM, Seoul National University, Korea

A recent ultra-large SSD (e.g., a 32-TB SSD) provides many benefits in building cost-efficient enterprise storage systems. Owing to its large capacity, however, when such SSDs fail in a RAID storage system, a long rebuild overhead is inevitable for RAID reconstruction that requires a huge amount of data copies among SSDs. Motivated by modern SSD failure characteristics, we propose a new recovery scheme, called *reparo*, for a RAID storage system with ultra-large SSDs. Unlike existing RAID recovery schemes, reparo repairs a failed SSD at the NAND die granularity without replacing it with a new SSD, thus avoiding most of the inter-SSD data copies during a RAID recovery step. When a NAND die of an SSD fails, reparo exploits a multi-core processor of the SSD controller in identifying failed LBAs from the failed NAND die and recovering data from the failed LBAs. Furthermore, reparo ensures no negative post-recovery impact on the performance and lifetime of the repaired SSD. Experimental results using 32-TB enterprise SSDs show that reparo can recover from a NAND die failure about 57 times faster than the existing rebuild method while little degradation on the SSD performance and lifetime is observed after recovery.

CCS Concepts: • **Hardware → External storage**; • **Computer systems organization → Secondary storage organization;**

Additional Key Words and Phrases: Die failure, ultra-large SSD, RAID, storage system

ACM Reference format:
Duwon Hong, Keonsoo Ha, Minseok Ko, Myoungjun Chun, Yoona Kim, Sungjin Lee, and Jihong Kim. 2021. Reparo: A Fast RAID Recovery Scheme for Ultra-large SSDs. *ACM Trans. Storage* 17, 3, Article 21 (August 2021), 24 pages.
https://doi.org/10.1145/3450977

1 INTRODUCTION

Ultra-large solid state drives (**UL SSDs**, e.g., 32-TB SSDs in a 2.5-inch form factor [1]) are becoming popular these days in enterprise storage markets because of their advantages in reducing the

This work was supported by Samsung Research Funding Incubation Center of Samsung Electronics, Republic of Korea under Project Number SRFC-IT2002-06. The ICT at Seoul National University provided research facilities for this study.
Authors' addresses: D. Hong, M. Chun, Y. Kim, and J. Kim (corresponding author), Seoul National University, 1, Gwanak-ro, Gwanak-gu, Seoul 08826, Korea; emails: {duwon.hong, mjchun, yoonakim, jihong}@davinci.snu.ac.kr; K. Ha and M. Ko, Samsung Electronics, 1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18448, Korea; emails: {keonsoo.ha, minseok2.ko}@samsung.com; S. Lee, Daegu Gyeongbuk Institute of Science and Technology (DGIST), 333, Techno jungang-daero, Hyeonpung-eup, Dalseong-gun, Daegu 42988, Korea; email: sungjin.lee@dgist.ac.kr.

1553-3077/2021/08-ART21 $15.00
https://doi.org/10.1145/3450977

total cost of ownership. As the capacity of a single SSD increases, fewer SSDs are needed to build a storage system. A smaller number of SSDs directly reduce various operating costs of storage systems, such as rack space, power, cooling, and storage-area networking costs.

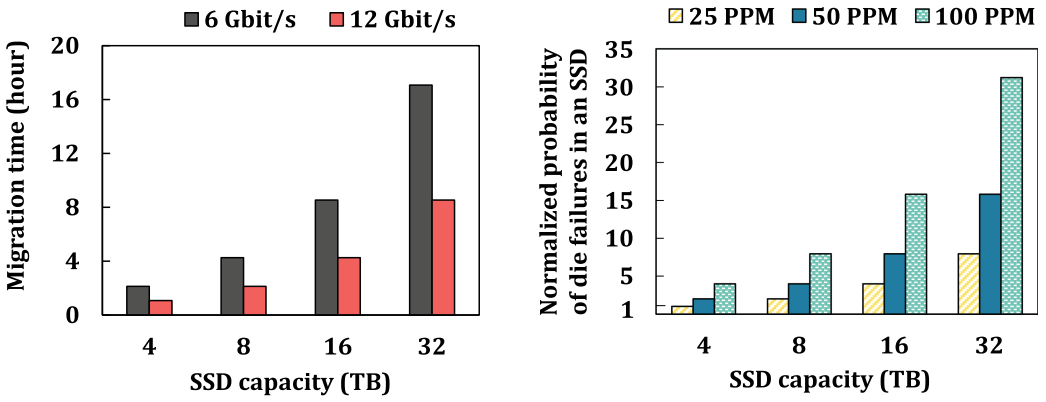
Ultra-large SSDs enable us to build cost-efficient storage systems, but high data recovery overheads become one of the key obstacles for their wider adoption in practice. For example, consider an enterprise storage system that consists of four UL SSDs grouped by **Redundant Array of Independent Disks-5** (**RAID-5**[1]) [2]. When one of SSDs in the RAID group fails, the failed SSD should be replaced by a new and expensive UL SSD. Then, a RAID recovery scheme should be triggered to reconstruct the RAID-5 organization. The RAID recovery process needs to read data and parity chunks from the working SSDs of the RAID group and write the recovered data to a new SSD. In general, the majority of the recovery time is proportional to the capacity of the failed SSD. Figure 1(a) illustrates the RAID recovery time when the SSD capacity changes under two different host interface speeds, 6 Gbit/s and 12 Gbit/s, of SAS-1 and SAS-2 protocols, respectively. Although all the I/O bandwidth of SSDs is assumed to be fully utilized for the rebuild process, the total recovery of UL SSDs takes a long time (e.g., 8.8 hours for a 32-TB SSD with 12-Gbit/s SAS-2). In practice, since only a portion of the I/O bandwidth can be used for a RAID rebuild process [3], the total recovery time may take a few days with a high possibility of the secondary disk failure.

Even worse, the frequency of UL SSD failures is expected to be increased as the number of NAND dies in an SSD increases. As the capacity of an SSD gets larger, more NAND dies are needed for an SSD. For example, sixty-four 512-Gib NAND dies are sufficient to build a 4-TB SSD but, for a 32-TB SSD, five hundred twelve 512-Gib dies [1] are required. A large number of NAND dies, however, significantly increases the possibility of die failures. Figure 1(b) shows how the probability of die failures in an SSD changes under varying SSD capacities with three different die defect rates.[2] Probability values of Figure 1(b) are normalized over the baseline case of a 4-TB SSD with the die defect rate of 25 PPM (parts per million). When the SSD capacity increases from 4 to 32 TB, the number of defective dies increases up to about 8 times and 31 times when the die defect rate is 25 PPM and 100 PPM, respectively. In existing SSD management schemes [4], a NAND die failure is considered as an entire SSD failure, because it generates many bad blocks. Thus, even a single NAND die failure results in the entire RAID reconstruction, requiring SSD replacement costs as well as long recovery time. The long RAID recovery process also increases the possibility of permanent data loss, because it is more likely that consecutive die/SSD failures would happen before finishing the recovery process (e.g., double disk failures in RAID 5 [5, 6]).

In this article, we claim that UL SSD failures should be handled at the SSD level first before taking place a data recovery process at the RAID level. Our proposal is motivated by two key observations. First, in contrast to small-sized SSDs whose capacity is few GB, it is feasible to repair failed NAND dies in UL SSDs. As pointed out earlier, UL SSDs are composed of many NAND dies (e.g., 512 dies for a 32-TB UL SSD). Thus, failures of few NAND dies do not badly affect the reliability of an entire SSD and can be normally operated. Moreover, by leveraging data redundancy in RAID, UL SSDs are able to recover data of failed dies, providing promised capacity with end-users. This self-recovery at a UL SSD level makes it possible to avoid the costly RAID reconstruction process as well as the hardware replacement. Second, a recent field study [6] that analyzes SSD failures in the enterprise storage system reports that there is a strong misconception on SSD failures. Most SSDs fail not because their flash cells were worn over their endurance limit but because they

[1]Although our technique has no dependence on the RAID type, we assume that the RAID 5 scheme is used for the description purpose. Where no confusion arises, we use RAID and RAID 5 interchangeably.

[2]We model the number X of failed dies in an SSD with n NAND dies as $X \sim B(n, p)$ where p is the probability of a single NAND die failure. That is, the probability P of die failures in the SSD is given by $P = \sum_{k=1}^{n} \binom{n}{k} p^k (1-p)^{n-k}$.



(a) Changes in recovery times. (b) Changes in die failures.

Fig. 1. Impact of UL SSDs on SSD recovery and failure.

experience unexpected component failures. Being the most dominant component of a UL SSD, NAND dies significantly contribute to such sudden failures in the UL SSD. Therefore, there is a strong incentive to devise a die-level SSD recovery scheme for UL SSDs.

We present a novel RAID recovery scheme, called reparo,[3] which repairs UL SSDs from a die failure through efficient on-line die rebuild techniques. To the best of our knowledge, reparo is the first technique that repairs a failed SSD at the die level. In order to minimize the time to recovery from a die failure, reparo minimizes both a die failure detection time and a rebuild time. Whenever a bad block is detected, reparo checks its neighboring blocks to detect a die failure early. Once a failed die is detected, multiple flash cores work in parallel to recover data in the failed die. Since a repaired SSD continues to be used, it is important for reparo to provide high performance and longer lifetime after the recovery. To this end, when rebuilding a failed die, reparo modifies a logical address-to-die mapping scheme in the way that minimizes space utilization imbalance among flash cores. This prevents performance degradation and lifetime drops that are caused by per-core space and workload variations.

To validate the proposed scheme, we have implemented reparo in Samsung PM1643 SSD [1], which supports up to 32 TB. Our experimental results show that reparo can recover from a die failure about 57 times faster than the existing rebuild method while little degradation on the SSD performance and lifetime are observed after recovery.

The rest of this article is organized as follows. We explain SSD failures focusing on their key causes and characteristics in Section 2. In Section 3, we present the key motivations behind reparo scheme. The proposed reparo scheme is described in Section 4 while its two optimization techniques are covered in Sections 5 and 6. Sections 7 and 8 describe our evaluation results and related work, respectively. We conclude in Section 9 with a summary and future work.

2 SSD FAILURES: CAUSES AND CHARACTERISTICS

2.1 SSD Failure Types

SSD failures can be grouped into two categories depending on its predictability [7]. The first category is the **end-of-life (EoL)** failure that is caused by worn-out SSD components. However, the

[3] Reparo is a charm used to repair a broken object from Harry Potter.

second category is the sudden failure, and it is highly associated with unexpected component failures that happen randomly.

Most SSD failures in the EoL failure category come from the worn-out NAND flash memory. Although the NAND flash memory is non-volatile, its reliable data retention is limited by the maximum number of **program/erase (P/E)** cycles that is determined by flash manufacturers. Recent three-dimensional (3D) TLC flash memory can support up to 10K P/E cycles. When flash cells in a block are worn out beyond their reliability threshold, the block becomes a bad block, because the pages in the block cannot be reliably accessed anymore [4, 8–10]. When the number of bad blocks in an SSD reaches the pre-defined maximum number N_{bad}^{ssd}, the SSD is considered to be failed. Since the number of bad blocks tends to increase rapidly around the end of SSD's useful life period, manufacturers set N_{bad}^{ssd} conservatively. For example, N_{bad}^{ssd} is typically set to 2.5% of the total (user-visible) blocks in an SSD [11].

The sudden failure happens abruptly at any point of time, so it often occurs much earlier than the EoL failure. The representative examples include NAND die failures and bugs in the **flash translation layer (FTL)**. A NAND die failure occurs when an excessive number of bad blocks are found in the NAND die, but there are various reasons that cause it. One example is when the peripheral circuitry (e.g., page buffer, **word line (WL)** decoder and sense amplifier) of a NAND die malfunctions because of some defects. In that case, all the blocks in the NAND die become bad, since they can no longer be reliably accessed. As another example, if the flash cell is not functioning properly due to a structural failure (e.g., WL to WL bridge, WL to channel bridge, and WL to common source line bridge), it is identified as a bad block during the manufacturing process or the infant period [12, 13]. However, if the number of accumulated bad blocks per NAND die is below a certain threshold, it is considered normal. The NAND specification defines the maximum number of bad blocks, N_{bad}^{die}, that can occur on a NAND die within its lifetime and if a NAND die exceeds the threshold, it is considered defective [11]. This is because the NAND manufacturer does not guarantee normal operations on these NAND dies. In fact, in the SSD field study, a large number of additional bad blocks tend to be generated in a short period of time after a certain number of bad blocks occur in most SSDs [4]. This is due to defective NAND dies with poor cell characteristics or defective peripheral circuits. Besides the excessive bad blocks, a defective NAND die can cause firmware operation failure or command timeout.

Since an SSD is tolerant for some number of bad blocks, it is important to efficiently manage bad blocks when bad blocks are detected during runtime. The bad block management module of the FTL remaps a bad block to a reserved block that comes from an **overprovisioning (OP)** space of the SSD. The OP space, which is a reserved space in the SSD, is used for minimizing the performance/lifetime impact of garbage collection and bad block management [14]. As the number of bad blocks increases, more blocks from the OP space are consumed to restore data of the bad blocks, which, in turn, negatively affects the performance and lifetime of the SSD.

2.2 SSD Failure Characteristics

SSD failure characteristics are commonly modeled using a typical bathtub curve [5] with three distinct periods: the infant period with high sudden failure rates, the useful life period with lower failure rates, and the wear-out period with high EoL failure rates. Since early failures are known to be quickly decreasing in most SSDs, most SSD reliability enhancement techniques have focused on extending the useful life period by better managing the flash wear-out speed. In particular, managing a NAND die failure was not the main focus of such techniques.

However, a recent field study on SSD failure characteristics in enterprise storage systems indicates that a typical bathtub model does not hold for SSD failures [6]. Redrawn from Reference [6],

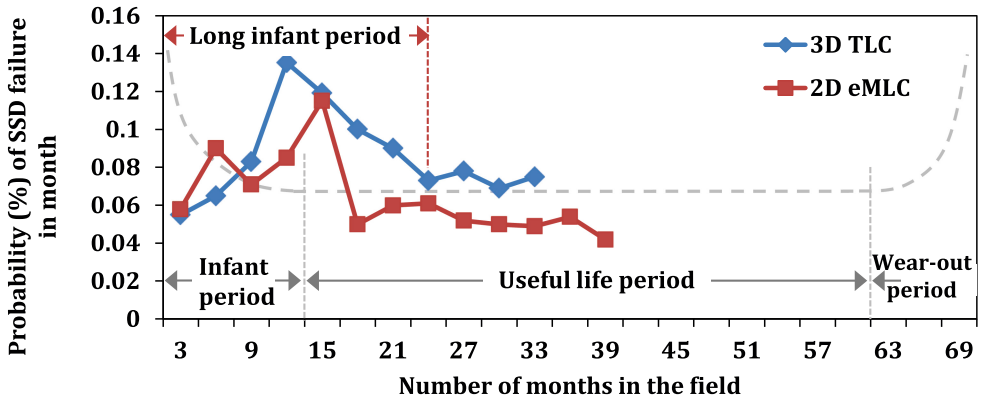


Fig. 2. Distributions of SSD failures over the SSD age [6].

Figure 2 summarizes probability distributions of SSD failures throughout the SSD lifetime for 3D-TLC SSDs and 2D-eMLC SSDs. Note that, as a reference case, a bathtub curve is also shown in a gray dotted curve. As shown in Figure 2, the SSD failure distribution is quite different from a typical bathtub model. The infant period was much longer than that of the bathtub model. Particularly, in 3D-TLC SSDs, the infant period with high failure rates lasted almost 2 years. Furthermore, there were no wear-out failures in most SSDs, because the number of P/E cycles performed did not exceed its limit even when SSDs have been used for several years. In fact, most SSD failures occurred within the useful life period.

The SSD failure trend reported by [6] strongly suggests that we should focus more on handling sudden SSD failures over EoL SSD failures. There are many reasons (e.g., firmware bugs) that result in sudden SSD failures, but the SSD capacity is expected to be the most significant factor that decides a sudden failure rate of UL SSDs. Since the number of NAND die failures increases linearly as the SSD capacity, it is more likely that the impact of NAND die failures becomes significant in UL SSDs. Therefore, a new RAID recovery scheme that focuses on die failures is strongly needed.

Another interesting observation in UL SSDs is that a NAND die failure is decoupled from an SSD failure. For example, in a 512-GB SSD with 8 NAND dies, 5,460 blocks become bad when one NAND die fails (assuming each die has 5,460 blocks). Since 5,460 bad blocks outnumber N_{bad}^{ssd}, a single die failure results in an SSD failure. However, in a 32-TB UL SSD, 5,460 bad blocks is only 0.2% of the total (user-visible) blocks whose number is much less than N_{bad}^{ssd}. If a failed die can be recovered, then UL SSD has a high potential to tolerate a die failure, preventing an SSD failure.

3 IMPACT OF UL SSDS ON RAID RELIABILITY

In this section, we explain the key motivations behind reparo using a hypothetical RAID-5 storage system, *UL-RAID(n)*, which employs n 32-TB UL SSDs [1]. Since the exact failure rate of a commercial UL SSD is not available, we assume that SSDs in *UL-RAID(n)* follow the SSD failure rates of 3D-TLC SSDs shown in Figure 2. Figure 3 shows the probability of RAID rebuilds in *UL-RAID(n)* with varying n's.[4] As the number of SSDs increases in *UL-RAID(n)*, more frequent RAID rebuilds are required. Since RAID rebuilds can interfere with normal host I/O requests, the increased number of RAID rebuilds badly affect user-perceived performance.

[4]We model the number X of failed SSD in *UL-RAID(n)* as $X \sim B(n, p)$ where p is the probability of an SSD failure. The probability of RAID rebuild, therefore, is given by $\sum_{k=1}^{n} \binom{n}{k} p^k (1-p)^{n-k}$.

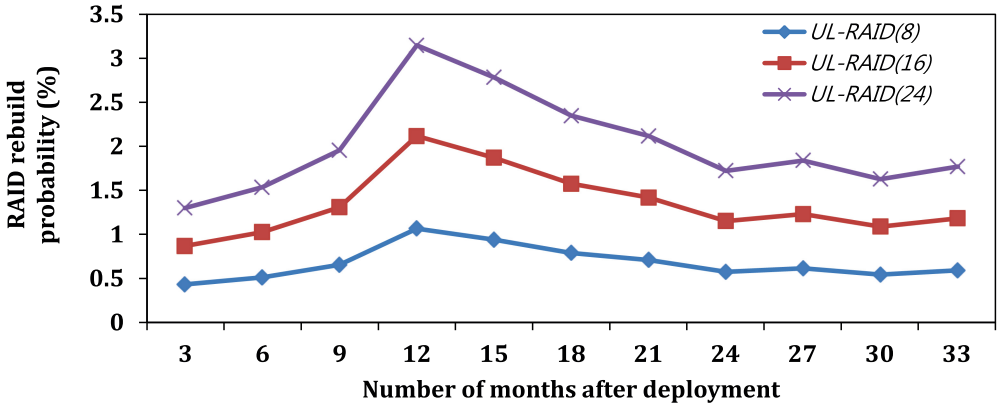


Fig. 3. Probability distributions of rebuilding a RAID group over the RAID group size.

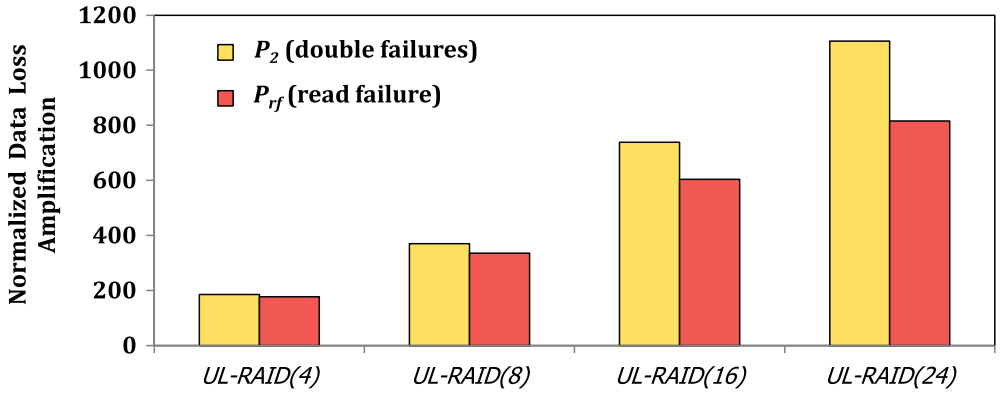


Fig. 4. Normalized data loss amplification in *UL-RAID(n)*.

Frequent RAID rebuilds also cause more critical reliability issues in *UL-RAID(n)*. First, the probability of deadly double disk failures [15] increases greatly. When the second SSD failure occurs during a RAID rebuild, the failed SSD cannot be recovered thus the user data loss is inevitable. The probability P_2 of a double SSD failure can be represented as follows:

$$P_2 = \frac{MTTR_{ssd} \times (n-1)}{MTTF_{ssd}}, \tag{1}$$

where $MTTR_{ssd}$ and $MTTF_{ssd}$ indicate the mean time to repair and the mean time to failure of an SSD, respectively. Since $MTTR_{ssd}$ increases linearly over the SSD capacity, P_2 increases linearly as well. For example, in *UL-RAID(16)*, P_2 increases 64 times over that in a RAID array with sixteen 512-GB SSDs.

Second, the probability of a read failure, P_{rf}, from a latent sector error during a rebuild can be increased. The probability of a read failure can be expressed as follows:

$$P_{rf} = 1 - (1 - UBER)^{S_{read}}, \tag{2}$$

where $UBER$ (uncorrectable bit error rate) is a fixed value (e.g., 10^{-16}) by SSD manufacturers and S_{read} is the total number of reads during a RAID rebuild. In *UL-RAID(n)*, P_{rf} is significantly increased as well. For example, in *UL-RAID(16)*, P_{rf} increases 52 times over that a RAID group with sixteen 512-GB SSDs.

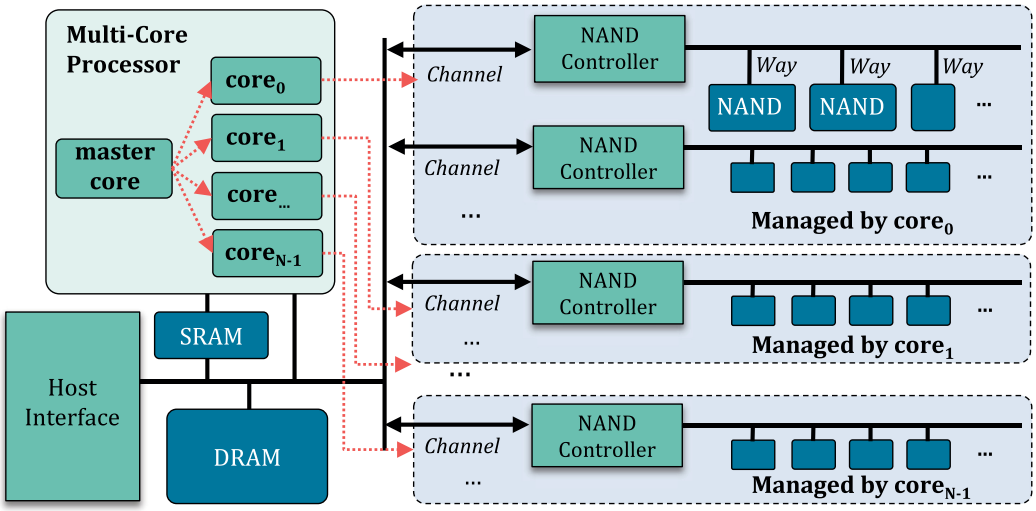


Fig. 5. An overall organization of an SSD architecture.

Figure 4 shows how much the probability of data loss from double failures and read failure amplifies as n increases in *UL-RAID(n)*. All the numbers are normalized over a RAID array with four 512-GB SSDs. Although we estimated the data loss probability conservatively by assuming that NAND die failures contribute only 3% of the sudden SSD failures in a 512-GB SSD, the data loss probability in *UL-RAID(n)* substantially increases as n increases. For example, data loss is more than 1,000 times likely in *UL-RAID(24)* over the baseline RAID with four 512-GB SSDs. The results in Figure 4 strongly indicate that *UL-RAID(n)* may not guarantee the same level of data reliability over when a RAID storage system was built using small-sized SSDs. Unless the reliability problem is resolved in an efficient fashion, employing *UL-RAID(n)* in real-world applications may not be practical in a near future.

4 RAID RECOVERY USING REPARO

4.1 Target UL SSDs

Since a UL SSD needs to support high performance for its huge storage space, a high performance multi-core processor is used for its SSD controller. Figure 5 shows an organizational overview of a typical UL SSD architecture. The SSD architecture consists of an $(N + 1)$-core multi-processor, DRAM/SRAM memory, a host interface logic, and a large number of NAND dies (which are grouped into different channels). For example, in Samsung enterprise SSD with 32-TB capacity, a quad-core ARM based processor is used to manage five hundred twelve 512-Gib dies, which are organized into 16 channels. In order to exploit the parallelism supported by the multi-core processor without a high management complexity, each core is often dedicated to handling a specific set of tasks. As shown in Figure 5, the master core is responsible for interfacing with the host system while the flash cores, $core_0, \ldots, core_{N-1}$, are assigned to flash management tasks. When the host system sends I/O requests to the UL SSD, the master core distributes the I/O requests across flash cores. To make flash management simpler, each flash core is dedicated to specific NAND dies. For example, when N flash cores are used, $\frac{1}{N}$ of the NAND dies are equally assigned to each flash core. Given a **logical block address (LBA)**, a simple address stripping method is used to decide a target flash core, $core_{target}$ (i.e., $core_{target} = LBA \bmod N$).

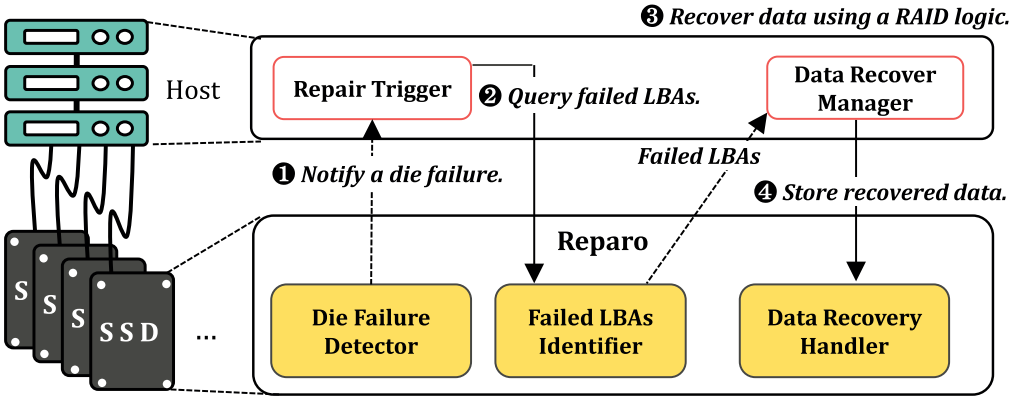


Fig. 6. An organizational overview of the reparo scheme.

4.2 Overview of Reparo

The proposed reparo scheme meets two requirements of a RAID recovery scheme for UL SSDs. Unlike the existing techniques, reparo repairs a failed die, not a failed SSD, by rebuilding the failed die using a RAID logic. Figure 6 shows an organizational overview of the reparo scheme. It consists of three new modules in a UL SSD: die failure detector (*Detector*), failed LBAs identifier (*Identifier*), and data recovery handler (*Handler*). When a die failure is detected by *Detector*, reparo notifies a die failure to the host system (❶). Then the host requests a list of LBAs that belong to the failed die (❷). *Identifier* finds the failed LBAs and sends them to the host, and the host recovers the data of the failed LBAs using a RAID logic (❸). When the recovered data are written back to the SSD (❹), *Handler* distributes the data to proper flash cores. When the host queries failed LBAs (❷), it limits the LBA search range (e.g., 64 MB) to control time/resource overheads. Therefore, a recovery sequence (i.e., ❷,❸, and ❹) is repeated until all the LBAs are covered.

In reparo, when a flash core $core_i$'s die fails, we call $core_i$ as the victim core and the rest of flash cores are called helper cores. When only the victim core is used for *Identifier* and *Handler*, we call it an **isolated die recovery (IDR)** scheme. (We denote this version of reparo by $\text{reparo}_{\text{IDR}}$.) If the helper cores, as well as the victim core, participate for *Identifier* and *Handler*, then it is called a **cooperative die recovery (CDR)** scheme. (Similarly, we denote this version of reparo by $\text{reparo}_{\text{CDR}}$.) We present $\text{reparo}_{\text{IDR}}$ in this section, and $\text{reparo}_{\text{CDR}}$ is described in Section 5.

4.2.1 Die Failure Detector. To minimize the impact of a failed die on the SSD performance, reparo detects a die failure as early as possible. When a bad block B is found from a die $\mathbb{D}$, *Detector* checks blocks physically adjacent to B by reading the first page of the blocks. If the read operation to the first page fails, then *Detector* proceeds to the next block. When the number of accumulated bad blocks in the die $\mathbb{D}$ exceeds a pre-defined maximum number N_{bad}^{die}, *Detector* labels the die $\mathbb{D}$ as a failed die. When a failed die is detected, reparo notifies a die failure to the host system using a well-known host-to-SSD interface (such as SMART [16, 17] or Check condition [18]). In our current *Detector* implementation, a die failure can be detected no later than dozens of milliseconds after the first bad block of a defective die is identified.

4.2.2 Failed LBAs Identifier. Once the host system is notified of a die failure, its recovery module asks reparo for the failed LBAs using the `get_lba_status` command that returns a list of failed

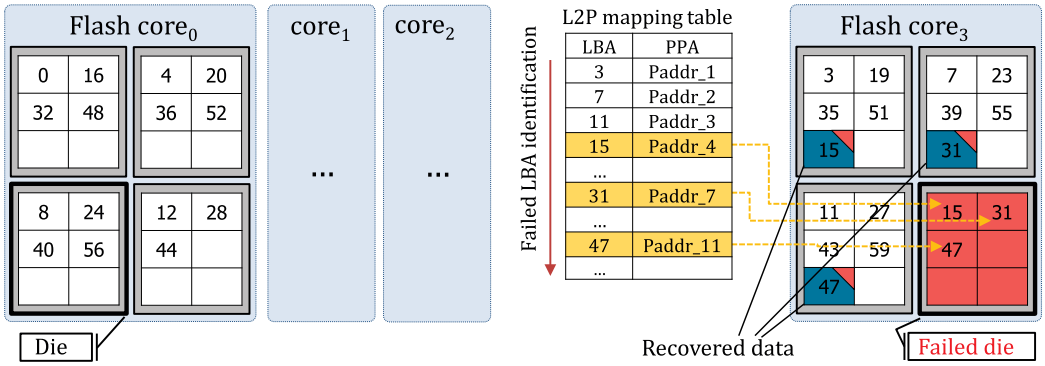


Fig. 7. An illustrative example of reparo$_{IDR}$.

LBAs from a specified LBA range $[l_{start}, \ldots, l_{end}]$.[5] If an FTL manages a separate **physical-to-logical (P2L)** mapping table, then *Identifier* can find failed LBAs of the pages in a failed die directly from the P2L mapping table using a physical address of a page in the failed die. Unfortunately, maintaining a P2L mapping table, in addition to an **logical-to-physical (L2P)** mapping table, is not feasible because of its large memory requirement. Typical FTLs manage only an L2P mapping table, which is essential for an FTL operation. Therefore, to respond to a failed LBA query, it is necessary to check the L2P mapping table for all LBAs in a query range to validate if they are stored in the failed die. For example, Figure 7 illustrates a case when a die failure occurs in one of the four dies managed by $core_3$. In reparo$_{IDR}$, $core_3$ becomes the victim core that is responsible for *Identifier* and *Handler*. When a query for failed LBAs in the LBA range of 0 to 15 is sent to the SSD, $core_3$ checks if any page in the requested LBA range is mapped to the failed die. Since LBA 15 is stored to a failed die, LBA 15 is sent to a host as a failed LBA. To identify all LBAs affected by a failed die, $core_3$ should check all the L2P mapping table entries of the SSD. In a UL SSD, since the number of L2P mapping entries is quite large, it is a key challenge to reduce the time overhead of *Identifier* in reparo.

4.2.3 Data Recovery Handler. After recovering the data of the failed LBAs through a RAID logic, a host stores the recovered data. When the victim core receives a write request, its *Handler* needs to store the recovered data to normal blocks. Since all the blocks in the failed die are bad blocks, *Handler* cannot use a conventional bad block management scheme that uses a reserved block in the same die (i.e., the failed die) for replacing a bad block. As a workaround, a reserved block of another die can be used to store the recovered data. However, this type of block remapping within the victim core complicates the LBA-to-die mapping, because multiple LBAs can be mapped to the same die. Instead of the remapping approach, in reparo$_{IDR}$, the victim core's *Handler* adjusts all the FTL steps so that they can work without the failed dies. For example, the die-stripping algorithm of the victim core is modified to use one less dies than before a die failure. In Figure 7, *Handler* of $core_3$ skips the failed die so that the recovered data of LBAs 15, 31, and 47 are evenly stripped and stored in the remaining dies.

Since only the victim core is used for *Handler* in reparo$_{IDR}$, the other cores work as if no die failure has occurred. For example, the master core does not need to change its static core mapping scheme while helper cores work for their allocated dies as usual. However, the victim core should

[5]This command is defined in the industry standard SCSI interface and has been extended to support Rebuild Assist [19] for a fast RAID rebuild.

Table 1. Changes in Space Utilization of IDR Scheme

	Number of dies per flash core					
	4	8	16	32	64	128
victim core (w/ die failure)	33.33%	14.29 %	6.67%	3.23 %	1.59%	0.79%
helper core	0%	0%	0%	0%	0%	0%

work with a reduced physical space after the recovery, which can significantly impact the overall SSD performance and lifetime.

5 COOPERATIVE DIE RECOVERY

Although $\text{reparo}_{\text{IDR}}$ of the previous section is simple to implement, $\text{reparo}_{\text{IDR}}$ can be further improved by relaxing its design constraint that only the victim core is involved in the recovery process. In this section, we describe $\text{reparo}_{\text{CDR}}$, which improves $\text{reparo}_{\text{IDR}}$ in two aspects by allowing all the flash cores to be utilized in parallel during the recovery process.

5.1 Identifier: Parallel Search of Failed LBAs

Scanning an entire L2P mapping table to find out LBAs belonging to failed dies is a time-consuming operation. Moreover, considering a huge logical address space of UL SSDs, such a scanning operation takes a significant amount of time. For example, it takes at least half an hour for one flash core of a 32-TB Samsung PM1643 to scan the whole L2P mapping table. Since the execution time of *Identifier* can be a bottleneck in the overall recovery process, in $\text{reparo}_{\text{CDR}}$, we modify *Identifier* so that all flash cores can participate in searching the failed LBAs.

To parallelize *Identifier*, we modified the data organization of the L2P mapping table. Since each flash core manages its own logical space, which is separated from the other flash cores, the existing L2P table is structured so that no mapping entry can be shared among different flash cores. In $\text{reparo}_{\text{CDR}}$, when `get_lba_status` command is issued to the victim core, the L2P mapping entries in the requested search range are first moved to the shared memory area that all the flash cores can access. The copied mapping entries are divided into N distinct regions so that all N flash cores can work in parallel.

5.2 Handler: Per-Core Space Utilization Adjustment

The main side effect of the simple $\text{reparo}_{\text{IDR}}$ scheme is that it negatively affects the space utilization of a victim core. The space utilization U_i of a flash core $core_i$, which is defined as a capacity ratio of the logical space to the physical space allocated to $core_i$, is a key SSD metric that is directly related to the performance and lifetime of SSDs. In general, the smaller U_i's, the higher (or the longer) the performance (or lifetime) of an SSD. When each flash core was initially allocated with equal, say x dies, if one of the dies allocated to the victim core $core_v$ fails, U_v increases by $[1/(x-1) \times 100]\%$ over helper cores. For example, when 16 dies are initially allocated to each flash core, U_v increases by 6.7%. Table 1 summarizes the increase in space utilization for a single die failure according to the number of dies per flash core. An increase in U_v can reduce the same amount of the SSD lifetime where sequential write workloads are dominant (as in modern data-intensive apps). A higher U_v also increases the **Write Amplification Factor (WAF)** value, because the reduced OP space needs more frequent GC invocations. For example, in our evaluation, we observed that when U_v increases by 6.7%, the WAF value can be increased by 166%, which may degrade the SSD lifetime and the performance almost by 60%. In $\text{reparo}_{\text{CDR}}$, we modify *Handler* so that the difference in space utilization among flash cores can be minimized.

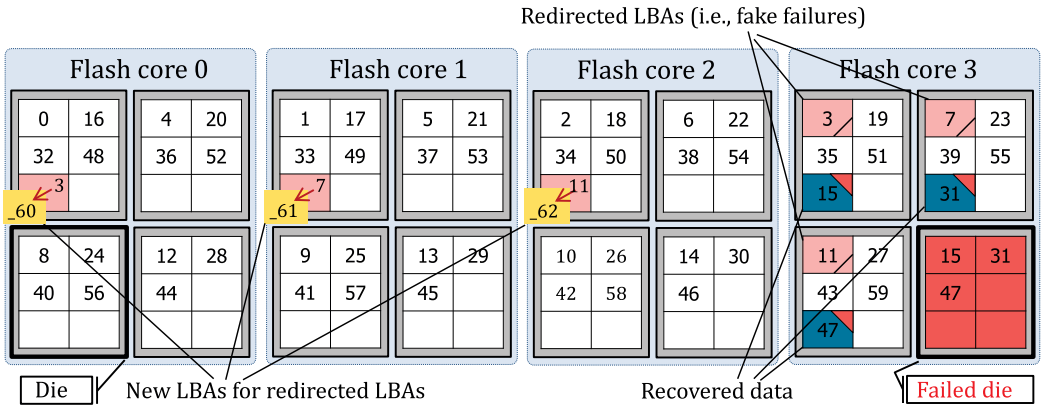


Fig. 8. An illustrative example for $\text{reparo}_{\text{CDR}}$.

5.2.1 Per-Core Logical Space Adjustment. To reduce the difference between U_v of a victim core $core_v$ and U_h of a helper core $core_h$, we reduce the capacity of logical space of $core_v$ while increasing that of $core_h$.[6] Assume that each flash core $core_i$ with D_i dies is allocated to the logical space LS_i where $LS_i \cap LS_j = \emptyset$ if $i \neq j$, and the capacity of LS_i is $|LS_i|$. We further assume that before a die failure, for all $0 \leq i \leq (N-1)$, 1) $|LS_i| = |LS|/N$ and 2) $D_i = D_{ssd}/N$ where D_{ssd} represents the total number of dies in an SSD[7]. To keep all U_i's equal after die failure recovery, the following equation should hold:

$$\frac{|LS_h| + \alpha}{D_{ssd}/N} = \frac{|LS_v| - (N-1) \times \alpha}{D_{ssd}/N - 1}, \tag{3}$$

where α represents the capacity of the extra logical space that should be added to each helper core. Solving Equation (3), α can be given as:

$$\alpha = \frac{|LS_h|}{D_{ssd} - 1}. \tag{4}$$

Consider an example scenario of a single die failure shown in Figure 8 where an SSD has four flash cores and each flash core has four dies ($N = 4$, $D_{ssd} = 16$ and $D_i = 4$). Assuming the capacity $|LS|$ of logical space of the SSD is 60, $|LS_i| = 15$ for all cores. Since the physical capacity of a NAND die is 6, initially, all U_i values are equal to 0.625 (i.e., 15/24). When a die fails from $core_3$, α is computed as $|LS_h|/15$, thus increasing the logical capacity of each helper core by 1 while decreasing the logical capacity of the victim core by 3. After this adjustment, all U_i values are still the same, but the space utilization has increased by 6.67% from 0.625 to 0.67. If there were no logical space adjustment, then the victim core's space utilization could increase by 33.3% to 0.83. Table 2 summarizes how space utilization changes after logical space adjustment for a single die failure under a varying number of dies per flash core. As expected, in both a victim core and a helper core, space utilization is increased by the same amount when a die fails. Furthermore, the increased amount of space utilization of the victim core is much lower compared to that of the IDR scheme because of the shared space adjustment among all cores.

[6]When no OP space becomes available for a helper core in $\text{reparo}_{\text{CDR}}$, a failed SSD cannot be repaired anymore. The failed SSD should be replaced by a new SSD in this case.

[7]Our technique can be generalized to a more general setting without these assumptions. However, because of a page limit, $\text{reparo}_{\text{CDR}}$ is presented under these assumptions.

Table 2. Changes in Space Utilization after the Adjustment

	Number of dies per flash core					
	4	8	16	32	64	128
victim core (w/ die failure)	6.67%	3.23%	1.59%	0.79%	0.39%	0.20%
helper core	6.67%	3.23%	1.59%	0.79%	0.39%	0.20%

Note that in the above description of the logical space adjustment technique, we assumed that all U_i values and D_i values were equal before a single die failure occurs. Since dies can fail more than once in an SSD, these assumptions generally do not hold and Equations (3) and (4) need to be modified. In case of multiple die failures, various failure combinations are possible. For example, multiple failures may be focused on a single flash core or they may be spread among multiple flash cores. Although treating an individual failure case using a case-specific equation will be the most accurate solution, we found that its management overhead can be substantial. Instead, we empirically evaluated the accuracy of space adjustment from Equation (4) in multiple die failures. Our evaluation results showed that the difference between the ideal adjustment solution and one from Equation (4) was negligible. For example, in the case of four die failures, the worst case for Equation (4) is when all four die failures occur in the same flash core. Even in this case, when each core has 64 or more dies (i.e., as in UL SSDs), α from Equation (4) was only 0.06% apart from the ideal adjustment value.

5.2.2 Selective LBA Redirections. To implement space utilization adjustment among flash cores as described above, we need to redirect $(N-1)\times\lfloor\alpha\rfloor$ LBAs from the victim core to $(N-1)$ helper cores while each helper core receives $\lfloor\alpha\rfloor$ additional LBAs for a die failure. Since the master core is responsible for distributing a host request to a proper flash core, all the redirection decisions are made at the master core without modifying how the flash cores work. The master core forms a unit of LBA redirections by $(D_{ssd}-1)$ consecutive LBAs that were originally mapped to the victim core. From each redirection unit, the master core selects $(N-1)$ LBAs and redirects the LBAs to helper cores one by one. Therefore, $(N-1)$ LBAs of $(D_{ssd}-1)$ LBAs of the victim core are redirected, effectively reducing its logical space by $(N-1)/(D_{ssd}-1)$. For example, in Figure 8, 15 LBAs form one redirection unit. Of 15 LBAs in one unit, 3 LBAs are redirected to $core_0$, $core_1$, and $core_2$.

To choose $(N-1)$ redirected LBAs from a redirection unit of $core_v$, the master core considers the first N LBAs from the redirection unit. Except for the vth LBA, the ith LBA is redirected to $core_i$. Formally, assume that the master core tries to select $(N-1)$ LBAs from a redirection unit $R=\{l_0,\ldots,l_{D_{ssd}-1}\}$ of $core_v$. Since each $l_i\in R$ can be expressed by $l_i=j\times N+v$ (where $(D_{ssd}-1)\times p\le j<(D_{ssd}-1)\times(p+1)$ for $p\ge 0$), $[j\,\%\,(D_{ssd}-1)]$ indicates the redirected core if it is not v and less than N. For example, in Figure 8, $R=\{3,7,11,15,19,\ldots,59\}$. First 3 LBAs, 3 $(=0\times 4+3)$, 7 $(=1\times 4+3)$, and 11 $(=2\times 4+3)$ are redirected to $core_0$, $core_1$, and $core_2$, respectively.

When an LBA is redirected to a helper core, the redirected LBA is stored in the extended LBA space $LS_{redirect}$ of the helper core that is hidden from the host system. Each flash core internally maintains its $LS_{redirect}$ area so that when the master core sends a request of a redirected LBA to its $LS_{redirect}$ area, it can be properly handled. To distinguish $LS_{redirect}$ from the host-visible logical space, we denote an LBA in $LS_{redirect}$ with the preceding underscore such as _60. In Figure 8, for example, three LBAs, 3, 7 and 11, of the victim core are redirected to three $LS_{redirect}$ LBAs, _60, _61, and _62, respectively.

5.2.3 Data Migration with Fake Failures. Since the master core changes its LBA-to-core mapping algorithm after a failed die is detected, if data are already written to the redirected LBAs of

the victim core, they should be moved to the newly redirected cores as well. However, since flash cores operate independently, direct data transfers between flash cores are difficult to implement. As an effective trick to this problem, we consider those redirected LBAs as failed LBAs although they do not belong to the failed die (i.e., the redirected LBAs are treated as fake failures.). In addition to the real failed LBAs, *Identifier* additionally searches fake failures and sends their LBAs to the host as well. When fake failures are recovered by the host, they are sent to the master core that then correctly redirects them to their new locations. Although this method incurs an additional RAID recovery cost, it is simple to be implemented as the existing recovery path is used. Furthermore, since the number of LBAs reported as fake failures is limited by the number of redirected LBAs (i.e., $(N-1) \times \alpha$), its overhead is not significant. For example, in Figure 8, three LBAs are reported as fake failures (because $\alpha = 1$ and $N = 4$).

6 IDENTIFIER ACCELERATION USING P2L MAPPING INFORMATION

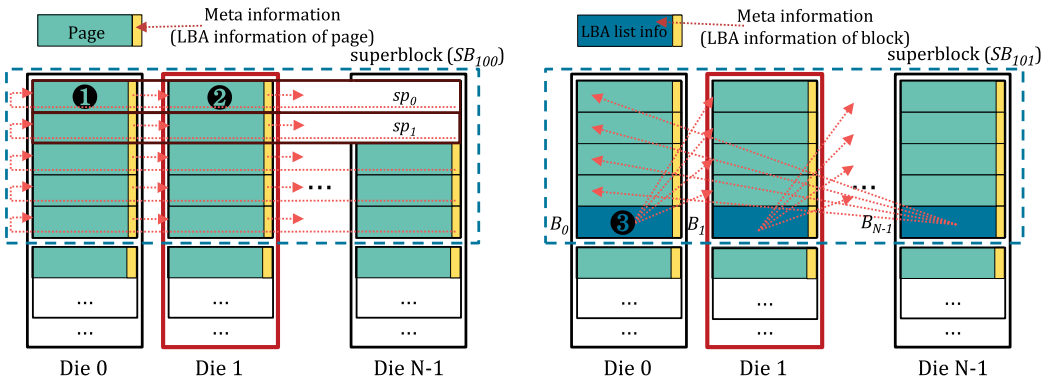
As mentioned in Section 4, SSDs do not generally manage P2L mapping information that can be used in identifying LBAs from a failed die. In this section, we propose two P2L management methods and *Identifier* acceleration techniques that can further reduce the key performance bottleneck of reparo.

6.1 Page-level P2L Entrustment to Neighboring Die

A NAND page consists of two areas, one for storing data and the other for storing an error-correction code and various FTL metadata. The latter area, called as the spare area of the NAND page, stores key information for operating an SSD reliably. One such information is the LBA of the data stored in the NAND page. The LBA information in the spare area is used when the mapping information of an SSD is reconstructed from unexpected failures (such as sudden power-off failures [20]) or when valid pages of the GC victim block are moved to different blocks. Therefore, it is possible to find out which LBAs are stored in a specific die by checking its spare area. However, when a die failure occurs, LBA information stored in the spare area of the failed die becomes inaccessible as well, thus making it impossible to read stored LBAs of pages in the failed die.

In order for the FTL metadata on the spare area to be available even when a die fails, the data page and its FTL metadata should be stored in different dies. In general, it is quite inefficient to store a page data and its metadata on two separate pages, because doing so requires two writes. However, in reparo, we propose a simple extension to a superblock-based mapping scheme [21, 22] so that a data page and its metadata can be stored without an extra write overhead on different pages in different dies. Unlike a page-level mapping scheme, the superblock-based mapping technique, which is widely used in practice, employs a superpage as a write unit. To support a superpage, a superblock is formed from k different blocks in k different dies. For example, Figure 9(a) shows that the superblock SB_{100} consists of N blocks from N different dies. A superpage of a superblock consists of k pages from k blocks (that are members of the superblock) where all k pages have the same page offset within their blocks. For example, in Figure 9(a), the superpage sp_0 consists of N pages that have the page offset of 0 within their blocks. In the superblock-based mapping technique, since a write to a superpage requires k writes to k pages (in different dies), we can easily separate data page and its metadata to two neighboring pages within the same superpage.

In the page-level P2L entrustment technique, we store data page and its metadata in two pages that are immediate neighbors within a given superpage. When a die fails, *Identifier* only needs to check the spare area of a neighboring die, instead of checking all the entries in the L2P mapping table. For example, as shown in Figure 9(a), when Die 1 fails, the failed LBAs of Die 1 can be identified from the spare area of the adjacent die, Die 0. The LBA information of the page ❷ in the superpage sp_0 can be obtained from the spare area of the page ❶ in Die 0.



(a) Page-level P2L entrustment. (b) Block-Level P2L entrustment.

Fig. 9. Illustrative examples of P2L entrustment.

6.2 Block-level P2L Entrustment to Neighboring Die

Although the page-level P2L entrustment technique can be efficiently implemented using the extended superblock-based mapping scheme, it incurs a significant overhead for *Identifier*, because all the pages of a neighboring die should be read. To identify failed LBAs more efficiently, we propose a block-level P2L entrustment technique that stores all the LBAs from a neighboring block in a single reserved page of a block, thus reducing the number of page reads per block by *Identifier* from the number of pages in a block to one. We use the last page of each block for this purpose.[8] Figure 9(b) illustrates how the block-level P2L entrustment scheme works. For a given superblock (e.g., SB_{101}), as with the page-level P2L entrustment scheme, each block's LBA information is stored in its neighboring block. However, in the block-level P2L entrustment technique, we dedicate the last page of each block for storing P2L mapping information of the neighboring block. When Die 1 fails, the failed LBAs of the block B_1 can be identified by reading the page ❸ of the block B_0. Since all the failed LBAs in a failed block can be found with a single page read, *Identifier* can work very efficiently. We denote an extended $\text{reparo}_{\text{CDR}}$ with the block-level P2L entrustment technique by $\text{reparo}_{\text{CDR}}{}^{*}$.

In the block-level P2L entrustment scheme, a single page should be able to contain all the LBA information of a block. Although recent NAND flash memory has a large number (e.g., 768) of pages in a block, a single page can easily meet this requirement. For example, consider a block with 768 pages where the page size is 16 KB. If we support a common 4 KB-based mapping scheme, then there are 3,072 mapping units in a block. When each mapping unit is referenced by a 32-bit address, all the LBA lists of a block can be stored on a single page (i.e., 12 KB < 16 KB). Furthermore, whenever a program operation is performed on a block, FTL needs to accumulate the list of all LBAs stored in each block in the buffer memory until all the pages in the block are programmed except for the last page. Since most FTLs limits the number of active blocks (where a requested page write is programmed) to 1 or 2 [23, 24], the memory requirement for buffering the LBAs of programmed pages is reasonable compared to the SSD capacity. For example, assuming that 2 active blocks per NAND die are maintained in a 32-TB SSD with 512 NAND dies, a 12-MB buffer memory is sufficient.

[8]To satisfy the sequential program constraint of the NAND flash memory, a superpage is sequentially written in a superblock. Therefore, it is logical to store the LBA list of a superblock to its last superpage.

6.3 Additional Considerations for P2L Entrustment

The proposed P2L entrustment schemes may not work when more than one die fail at the same time. For example, if two neighboring dies fail together, failed LBAs of one die cannot be identified. In this article, we assume that multiple die failures are possible but they do not occur at the same time. Since a die failure is a rare event and each die is physically independent of the other die, it is a reasonable assumption in practice.

When failed LBAs are identified by reparo$_{CDR}$*, they are not properly sorted, because when they were stored to the last page of a block, they were not sorted. However, when a host queries failed LBAs after a die failure is detected (as shown in ❷ of Figure 6), the host asks reparo of a list of failed LBAs within a specific LBA range. A naive solution would be to sort identified failed LBAs before reparo responds to the host query. However, sorting a large number of failed LBAs can be time-consuming. In the current implementation, when the failed LBAs are decided by reparo$_{CDR}$*, their L2P mapping entries are marked as failed LBAs. With a simple modification to the L2P mapping table, when the host queries for failed LBAs from a specific LBA range, reparo$_{CDR}$* can quickly identify the failed LBAs without a costly sorting step.

7 EXPERIMENTAL RESULTS

7.1 Experimental Settings

To evaluate the effectiveness of the proposed reparo schemes, we implemented the proposed schemes in Samsung PM1643 SSD [1], which can be configured for a 4-TB SSD (with 64 dies) and a 32-TB SSD (with 512 dies). The SSD controller of a PM1643 SSD consists of four flash cores along with one master core (as described in Section 4.1). Since there are four flash cores, each flash core handles one quarter of the NAND dies in the SSD. We set the initial space utilization of each core to 0.9. We assume a storage system with 8 SSDs configured in RAID 5. To emulate die failures during runtime, we added a special command that imitates a real die failure to PM1643 firmware. The special command, which was implemented as a vendor-specific command of SCSI [18], makes all NAND operations fail to a selected NAND die.[9] To simulate a die failure, this special command was requested from a test software (e.g., DriveMaster [25]).

We compared four techniques: baseline, reparo$_{IDR}$, reparo$_{CDR}$, and reparo$_{CDR}$*. The baseline scheme, which represents a state-of-the-art RAID recovery technique, is based on Rebuild Assist [19], which was proposed for a fast RAID recovery. In baseline, when a die failure occurs, it is considered as an SSD failure, and it replaces the failed SSD with a reserved SSD by copying all the valid pages in the failed SSD to the reserved SSD. During an SSD rebuild, baseline directly copies readable pages of the failed SSD using Rebuild Assist [19]. Reparo$_{CDR}$* works in the same way as reparo$_{CDR}$ except that its *Identifier* is optimized using the block-level P2L entrustment technique.

To evaluate four techniques, we have used two benchmark suites: FIO benchmark [26] and Iometer benchmark [27]. Using simple synthetic workloads (e.g., sequential read/write and random read/write) from the FIO benchmark, we compare how key steps of the recovery process work differently in the proposed schemes. To understand the effect of the proposed schemes in real-world settings, we used five enterprise application workloads [28] from the Iometer benchmark. Table 3 summarizes key I/O characteristics of these workloads.

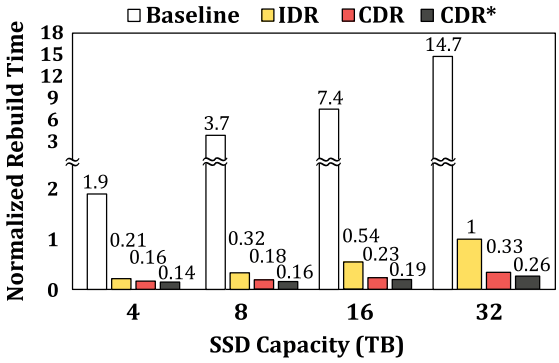
7.2 Experimental Results

7.2.1 Recovery Overhead. To compare the recovery overhead, we measured the rebuild time of each scheme in case of a die failure. Figure 10(a) compares normalized rebuild times of four

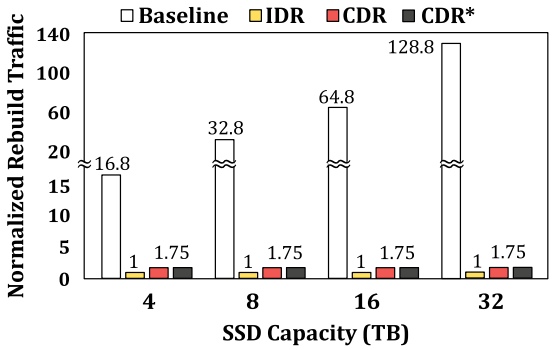
[9] In the current implementation, we modified the NAND flash reliability parameters (e.g., reference voltages) so that normal operations cannot be performed.

Table 3. I/O Workload Characteristics of Five Enterprise Applications

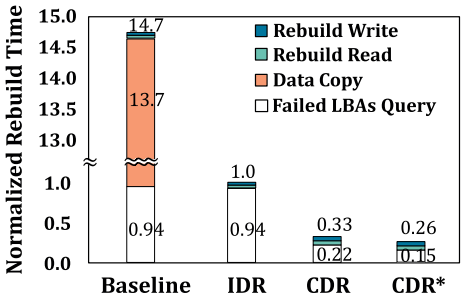
Application	Transfer Size	Read %	Write %	Random %	Sequential %
Media Streaming	64 K	98	2	0	100
File Servers	8 K	90	10	75	25
Database OLTP	8 K	70	30	100	0
Archive	2 M	55	45	95	5
Medical Imaging	1 M	5	95	5	95



(a) Rebuild time comparison.



(b) Rebuild traffic comparison.



(c) A rebuild time breakdown of 32-TB SSD.

Fig. 10. Comparisons of rebuild overhead.

techniques under varying SSD capacity. All the measurements were normalized over the result of $reparo_{IDR}$ for a 32-TB SSD. $Reparo_{IDR}$ completes the recovery process about ~9.3–14.7 times faster than baseline by minimizing data migration. $Reparo_{CDR}$ is about 3.1 times faster over $reparo_{IDR}$ because of its parallel *Identifier* module while $reparo_{CDR}$* is about 1.27 times faster over $reparo_{CDR}$ with its P2L entrustment support. Overall, $reparo_{CDR}$* is 57 times faster than baseline in a 32-TB SSD, even when the full I/O bandwidth of a host system is used for the RAID recovery. If the I/O bandwidth of RAID recovery is limited (e.g., only 10% of the total I/O bandwidth for the host) to minimize the impact of the RAID recovery on the performance of host request processing, then $reparo_{CDR}$* is about 110 times faster than baseline, which takes more than 3 days in a 32-TB SSD. Note that the rebuild time of reparo techniques very slowly increases over baseline when the SSD capacity increases. This observation illustrates the advantage of the reparo schemes that repair a single die instead of rebuilding an SSD as in baseline.

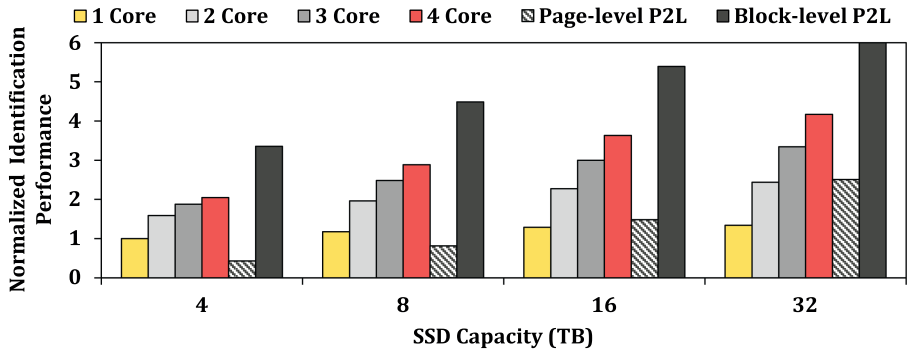


Fig. 11. Comparisons of *Identifier* performance.

Figure 10(b) compares four techniques in terms of the total amount of data movements during the rebuild time. Compared to baseline, the reparo schemes require up to two orders of magnitude fewer data movements during the recovery. Unlike baseline, the reparo schemes generate the same amount of rebuild traffic during the recovery regardless of the capacity of SSDs, because they rebuild a failed die only. Although $\text{reparo}_{\text{CDR}}$ moves more data over $\text{reparo}_{\text{IDR}}$ because of logical space adjustment, the overall rebuild time of $\text{reparo}_{\text{CDR}}$ is smaller than $\text{reparo}_{\text{IDR}}$ because $\text{reparo}_{\text{CDR}}$ significantly reduces time for finding failed LBAs by parallel query processing. Figure 10(c) shows a detailed breakdown of the total rebuild time during the rebuild process in a 32-TB SSD. As shown in $\text{reparo}_{\text{IDR}}$, the time taken by *Identifier* is the dominant factor of the total recovery time. The parallel *Identifier* of $\text{reparo}_{\text{CDR}}$ reduces the *Identifier* execution time by 76% over $\text{reparo}_{\text{IDR}}$. $\text{Reparo}_{\text{CDR}}{}^{*}$ further reduces the *Identifier* execution time by 31% over $\text{reparo}_{\text{CDR}}$ by the P2L entrustment technique.

To better understand the effect of various *Identifier* optimization techniques, we compared the performance of the failed LBA identification step in detail under varying number of flash cores participating in the parallel search and different P2L entrustment techniques. Figure 11 shows the normalized performance of the failed LBAs identification step for different SSD capacities. (The identification performance represents the processing speed measured from a host after a failed LBA query was sent to an SSD. The higher the identification performance, the shorter *Identifier* takes.) All the values were normalized over the query processing speed on a 4-TB SSD when a single core was used. As the capacity of the SSD increases, the identification speed tends to increase. This tendency is related to how often failed LBAs appear. For example, on a 4-TB SSD, on average, one of the 16 LBAs in the victim core is stored on the failed die, whereas on a 32-TB SSD, one of the 128 LBAs on the victim core is stored on the failed die. Whenever a failed LBA is identified, an internal data structure update is required to transmit the information to the host, and the mapping information related to the failed LBA needs to be updated. This is why the higher the frequency of failed LBAs, the slower the query processing speed. The high frequency of failed LBAs also gives a negative effect on the efficiency of parallel processing. Therefore, on a 4-TB SSD, the performance of the 4-core parallel search is about twice that of the single-core search. However, on a 32-TB SSD, the performance of the 4-core parallel search is 3.1 times higher than that of the single-core search.

As shown in Figure 11 the page-level P2L entrustment technique is quite slow, because it needs to read a large number of pages to identify failed LBAs from neighboring pages. For example, it takes longer than the 4-core parallel search case in all four SSDs. However, the block-level P2L entrustment technique can identify failed LBAs much faster than the 4-core parallel search on all

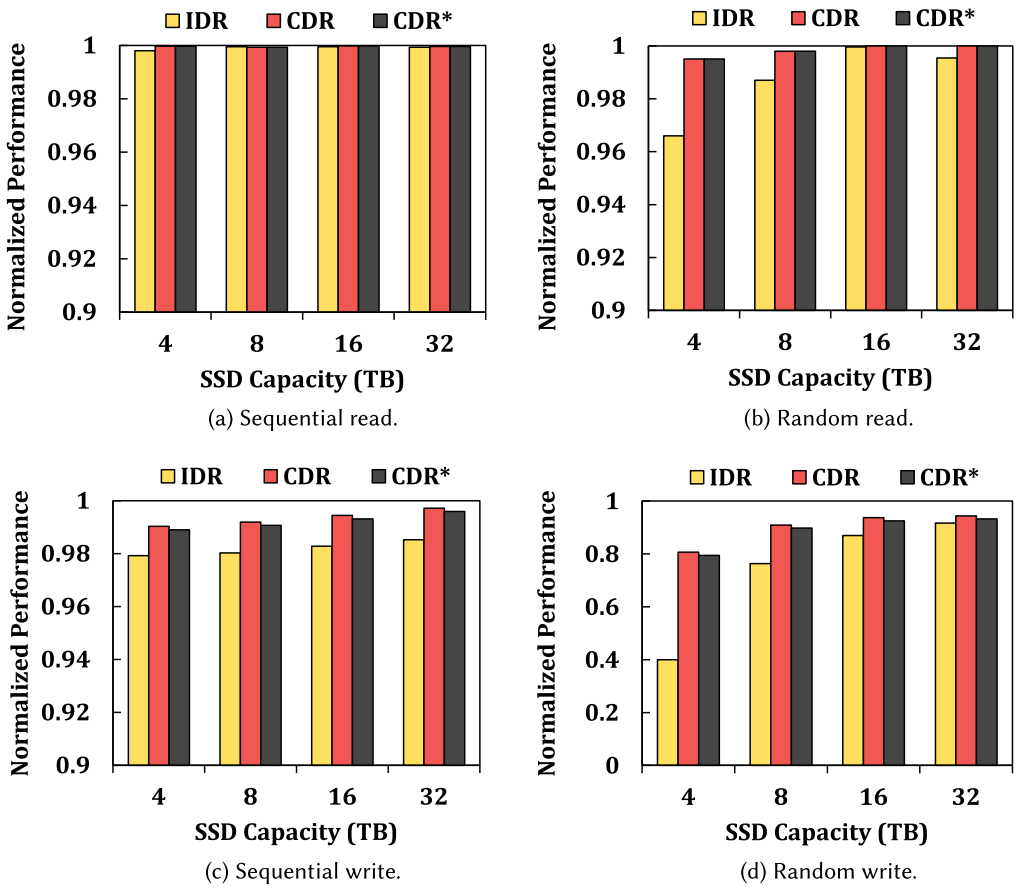


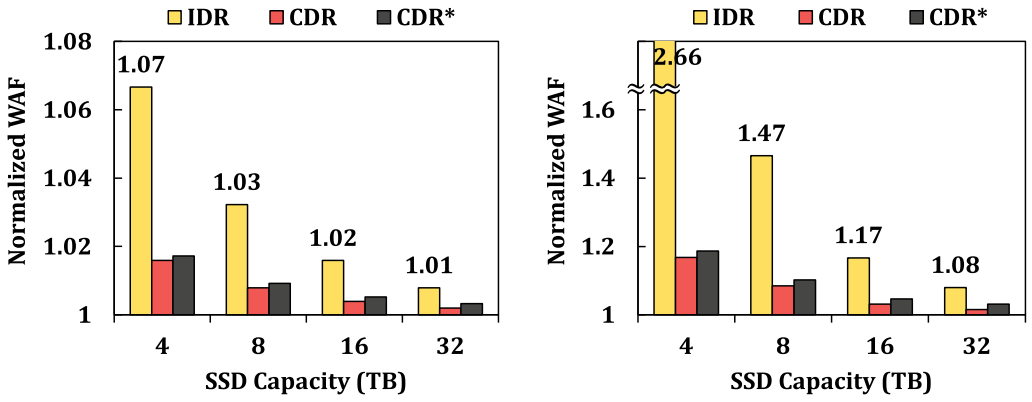
Fig. 12. Performance comparisons after die failure recovery.

SSD capacities. For example, on a 16-TB SSD and a 32-TB SSD, the block-level P2L entrustment technique outperforms the 4-core parallel search by 48% and 44%, respectively.

7.2.2 Post-recovery Performance/WAF Impact. To understand how a reparo-repaired SSD behaves in performance and lifetime aspects, we measured IOPS and WAF values after a single die failure was repaired by the reparo schemes. All the measurements are normalized over those of a new SSD without any die failure. We used four representative synthetic workloads generated through FIO.

Figure 12 compares IOPS values between the reparo schemes under four workloads. For the sequential read and random read workloads shown in Figure 12(a) and (b), different reparo schemes show almost the same performance as the new SSD. $\text{Reparo}_{\text{IDR}}$ has a performance drop up to 3.4% for random read under low-capacity conditions when the SSD capacity is relatively low (i.e., 4 TB), but the performance degradation is marginal in the other conditions. This is because the read bandwidth of the NAND flash is sufficient to satisfy the host interface speed even if the number of dies is reduced.

However, for write workloads, there is a little more performance differentiation among the proposed schemes although their difference is not significant. As shown in Figure 12(c), in the case of sequential write workload, small performance differences are largely from the reduced



(a) Sequential write workload. (b) Random write workload.

Fig. 13. Impact of die repairs on WAF.

NAND parallelism. For example, reparo$_{IDR}$ is 1.1% worse than reparo$_{CDR}$ in a 4-TB SSD, because its NAND parallelism is affected by the isolated die recovery scheme. In the random write workload, as shown in Figure 12(d), the performance difference between reparo$_{IDR}$ and reparo$_{CDR}$ is substantial, because the efficiency of GC is combined with the proposed schemes. Reparo$_{CDR}$ outperforms reparo$_{IDR}$ from 3% to 102%. This is mainly because reparo$_{CDR}$ better manages the performance-critical space utilization of flash cores over reparo$_{IDR}$. Imbalanced space utilization in reparo$_{IDR}$ directly affects the efficiency of GC, which, in turn, significantly degrades the overall IOPS. Reparo$_{CDR}$* shows almost the same performance as reparo$_{CDR}$, with only a performance drop of less than 1.5%, because reparo$_{CDR}$* uses the last page of each block for storing a list of LBAs, which reduces the capacity of the OP space slightly.

Figure 13 compares WAF values between the reparo schemes under two different write workloads. WAF values are normalized over the baseline case where no die failure occurs in a given SSD capacity. For the sequential write workload, as shown in Figure 13(a), WAF values do not increase significantly. Except for reparo$_{IDR}$ whose WAF value increased up to 7% at a 4-TB SSD, all the other schemes increased their WAF values by less than 2% over four different SSDs with different capacities. However, as shown in Figure 13(b), in the random write workload, there are larger variations in WAF values among different schemes in different SSDs. For example, the WAF value increases by 47% in reparo$_{IDR}$ for an 8-TB SSD while the WAF value increases less than 9% in reparo$_{CDR}$ for the same SSD. Large differences in Figure 13(b) come from the efficiency of GC in each scheme, because each scheme affects differently on the available OP space. As shown in Figure 13(b), reparo$_{CDR}$ is the most efficient in maintaining the available OP space evenly among flash cores. Reparo$_{CDR}$*, which has a smaller effective OP space than reparo$_{CDR}$, results in an additional 1.5% increased WAF value.

Figure 14 compares the performance of each scheme after a die failure is recovered using enterprise workloads from the Iometer benchmark. All the measurements were normalized over the SSD performance of the same capacity with no die failure. As expected, there are large differences among reparo$_{IDR}$ and reparo$_{CDR}$ on a small SSD when workloads are write-intensive. For example, in Archives and Medical Imaging workloads with high write request ratios, reparo$_{CDR}$ outperformed reparo$_{IDR}$ by 96.4% and 99.5%, respectively, on a 4-TB SSD. Even for write-intensive workloads, if the access pattern is sequential, the influence of the OP space can be low. However, when the random and sequential patterns are mixed, the influence of the OP space is large, similar to the 100% random pattern even if the random ratio is low. However, in read-intensive workloads,

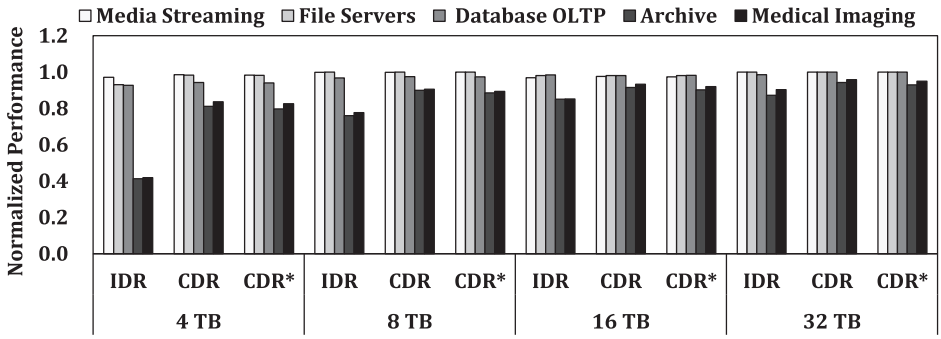


Fig. 14. Performance comparisons after die failure recovery with enterprise workloads.

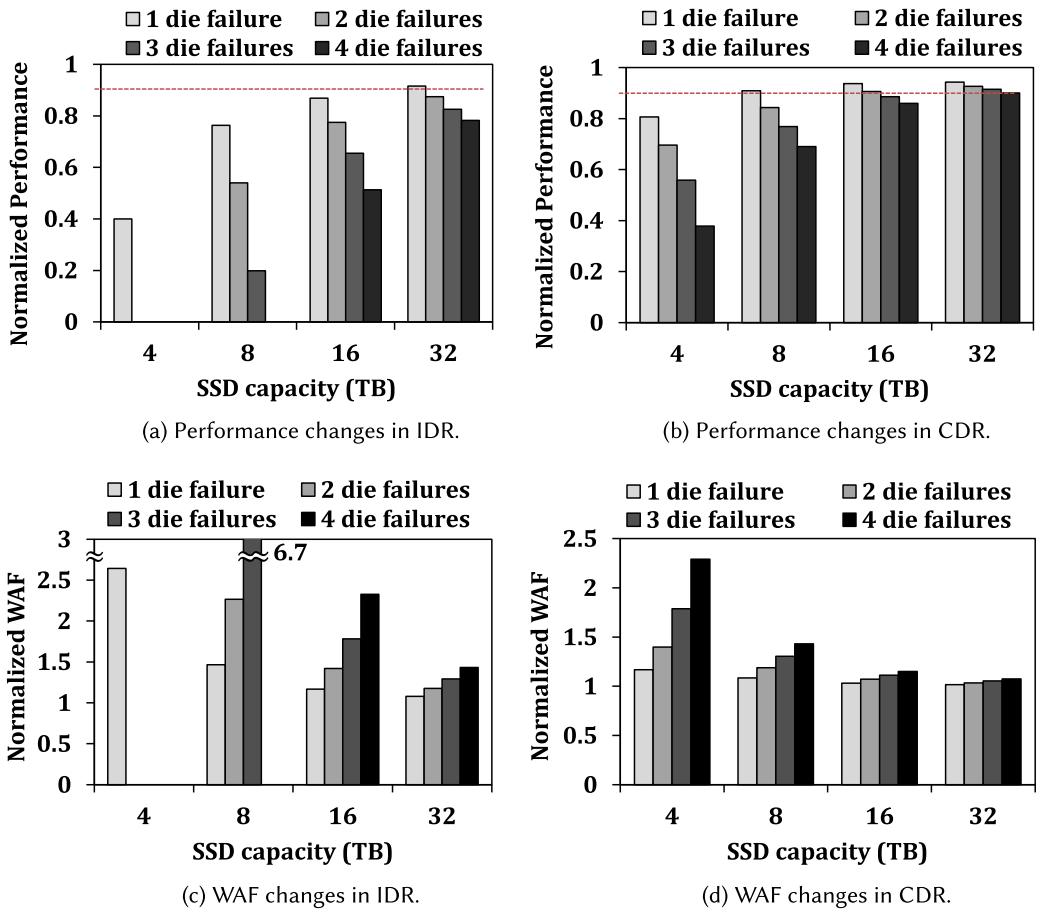
there is not much difference in performance for each scheme, which is in line with the evaluation results of Figure 12(a) and (b).

Note that we did not directly compare the lifetime impact of each scheme. However, since the total amount of written data to an SSD can be a useful indicator of the SSD lifetime, we can estimate the lifetime impact of each scheme using its WAF value w. Given a fixed amount W_{host} of host writes, the total amount T_{data} of written data to the SSD can be computed by $T_{data} \times w$. Using WAF values of Figure 13, for example, we can estimate that $\text{reparo}_{\text{IDR}}$ can decrease the SSD lifetime over when no die fails, by 2.9% and 31.9%, respectively, for sequential workload and random workload, on an 8-TB SSD after a single die is repaired. For the same conditions, $\text{reparo}_{\text{CDR}}$ shows a longer SSD lifetime, reducing the SSD lifetime by 0.8% and 7.8% only.

7.2.3 Post-recovery Impact of Multi-die Failures. Unlike when a single die failure is repaired, $\text{reparo}_{\text{IDR}}$ and $\text{reparo}_{\text{CDR}}$ shows significant differences in the post-recovery performance/WAF when multiple die failures are repaired. Figure 15 shows how performance and WAF changes under the random write workload as the number of failed dies increases assuming that all die failures occur in the same flash core, considering the worst case. As shown in Figure 15(a) and (c), when an SSD is repaired by $\text{reparo}_{\text{IDR}}$, its performance is quickly degraded while its WAF is rapidly increased. However, as shown in Figure 15(b) and (d), when an SSD is repaired by $\text{reparo}_{\text{CDR}}$, its performance is much slowly degraded as with the WAF increase.

Slow performance degradation of $\text{reparo}_{\text{CDR}}$ can be an important advantage in enterprise storage systems. In such systems, to support the sustained RAID performance [29], each SSD has a strict requirement on the performance degradation (such as the maximum 10% performance drop) [30]. For example, when the maximum performance drop is set to 10%, $\text{reparo}_{\text{CDR}}$ can survive up to four die failures in a 32-TB SSD. However, $\text{reparo}_{\text{IDR}}$ can only handle a single die failure under the same condition.

7.2.4 Sensitivity for the Space Utilization. Since die failure recovery is performed by utilizing the OP space, the available size of the OP space affects the performance and WAF after die failure recovery process. If the space utilization of an SSD is low, then more OP space is available and the performance impact by using reparo is reduced. To validate this observation, we measured performance and WAF while varying the space utilization from 0.9 to 0.87. Figure 16 shows how performance changes for successive die failures under different space utilization ratios. When space utilization is low, the performance degradation is also slowed accordingly. For example, in a 32-TB SSD, when the space utilization ratio is 0.87, $\text{reparo}_{\text{CDR}}$ can repair up to five die failures with a less than 10% performance degradation over four die repairs when the space utilization ratio is 0.90.



(a) Performance changes in IDR.

(b) Performance changes in CDR.

(c) WAF changes in IDR.

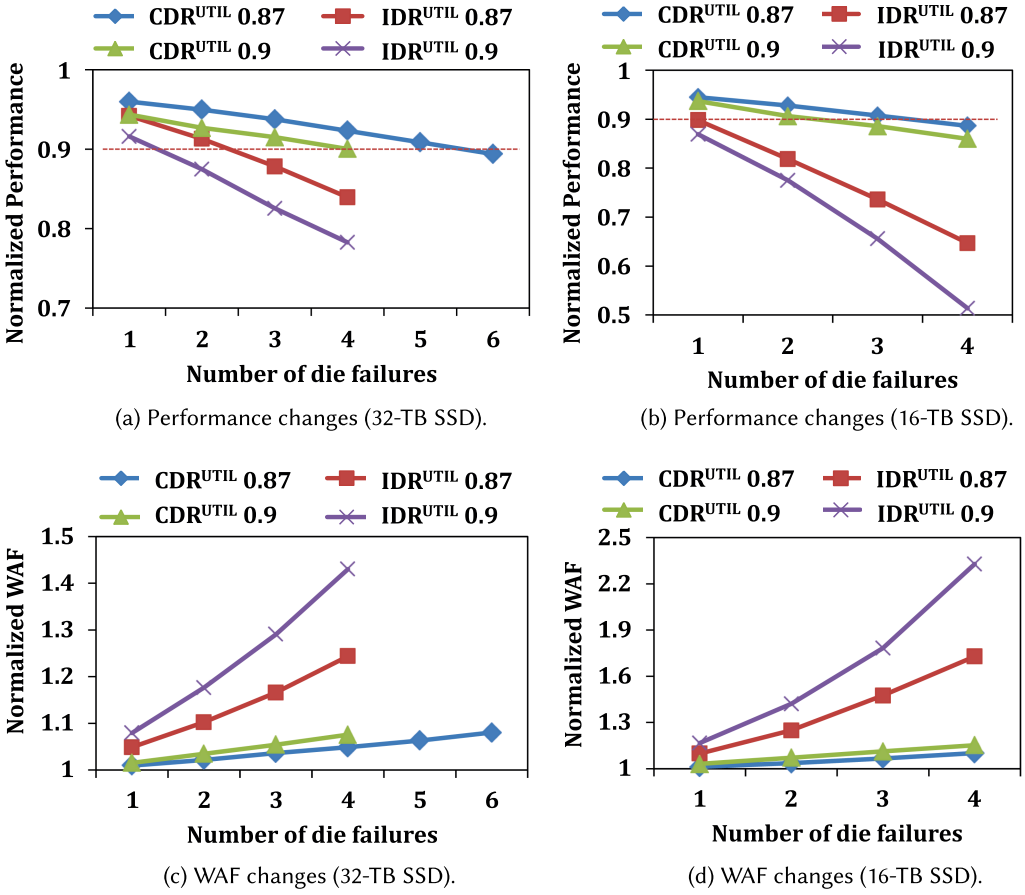
(d) WAF changes in CDR.

Fig. 15. Impact of multi-die failures on performance and WAF.

8 RELATED WORK

Fast RAID recovery techniques have been extensively investigated in enterprise storage systems [31–36]. For example, several groups have focused on devising efficient data layout methods that can reduce the impact of a RAID rebuild process on normal I/O requests from a host. The parity declustering layout was proposed by Muntz and Lui [31] to shorten rebuild time and improve user response by minimizing the number of disks required for reconstructing a failed disk so that the rest of disks can continue to handle host requests. Wan et al. [35] proposed a skewed sub-array organization in a RAID structure, which splits large disks into small logical disks to form sub-arrays but are configured to be skewed among physical disks. This enables a RAID rebuild process to be performed on multiple physical disks in parallel without access conflicts.

Although these schemes can reduce the total rebuild time by intelligently overlapping rebuild operations with host request processing, they do not reduce the total amount of data that need to be read for a RAID reconstruction task. Rebuild Assist [19] takes a different approach to expedite the RAID rebuild process. When an SSD fails, Rebuild Assist distinguishes the failed LBAs from the readable LBAs in the failed SSD. For the latter, Rebuild assist simply copies their data from the failed SSD to a replacement SSD without rebuilding them using a RAID scheme, thus reducing data reads from the rest of RAID storage. Reparo, which is based on a subset of new commands



(a) Performance changes (32-TB SSD).

(b) Performance changes (16-TB SSD).

(c) WAF changes (32-TB SSD).

(d) WAF changes (16-TB SSD).

Fig. 16. Impact of the space utilization on performance and WAF.

proposed by Rebuild Assist, is fundamentally different from Rebuild Assist in that reparo does not build a replacement SSD but repairs failed dies.

Reparo is similar to **Redundant Array of Independent NAND (RAIN)** techniques [37–43] in that they can recover a failed die inside a failed SSD. However, the existing RAIN techniques work at the individual SSD level rather than the RAID storage level, making them very difficult to use efficiently in a RAID storage system. For example, when a storage system is consist of RAIN-enabled SSDs, if an SSD fails (although such an SSD failure is much less likely because of an internal RAID configuration in a RAIN-enabled SSD configuration), its RAID recovery procedure will be as slow as that of a RAID storage system with normal SSDs. Since RAID should be supported in an individual SSD, the existing RAIN techniques incur a significant resource overhead (e.g., OP space reduction) as well as a flash lifetime degradation [44]. Therefore, the performance/lifetime of RAIN-enabled SSDs is poorer than SSDs without RAIN support. Furthermore, when a RAIN-enabled SSD is recovered after a die failure using a RAIN scheme, the OP space of the RAIN-enabled SSD will be further reduced, thus quickly degrading the performance/lifetime of the SSD. Since we are interested in continuing a normal operation of a RAID storage system after a failed die is recovered without replacing a failed SSD, we did not consider the RAIN techniques as a viable alternative solution for repairing failed dies. However, reparo, which was proposed for a

RAID recovery purpose, imposes little overhead on an individual SSD level while it can minimize the impact of the die-level recovery on the performance and lifetime of the repaired SSD.

9 CONCLUSIONS

We have presented a new RAID recovery scheme, reparo, to reduce the RAID recovery overhead from die failures in UL SSDs. Based on our key insight that a failed die of a UL SSD should be rebuilt at the die level (instead of the expensive and time-consuming SSD level), reparo, which employs the cooperative die recovery scheme, quickly recovers from a die failure and ensures no negative post-recovery performance/lifetime degradation on the repaired SSD. We have implemented reparo in a commercial enterprise SSD to validate its effectiveness. Our experimental results show that reparo can recover from a die failure about 57 times faster than the existing rebuild method while little degradation on the SSD performance and lifetime is observed after recovery.

The current version of reparo can be further improved in several directions. For example, if a die failure can be predicted with a high probability before it actually occurs, a die failure can be handled more effectively. Building such a predictor would be an interesting future direction if it can be possibly combined with a data-driven machine learning approach. In addition, we are also interested in improving the data transfer mechanism during the recovery process. For example, if data migrations between flash cores could be performed inside an SSD, unnecessary data transfers between the SSD and host can be reduced during the recovery process.

REFERENCES

[1] Samsung SSD. 2018. Retrieved from https://www.samsung.com/semiconductor/insights/news-events/samsung-starts-producing-industrys-largest-capacity-ssd/.

[2] David Patterson, Garth Gibson, and Randy Katz. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM-SIGMOD International Conference on the Management of Data.*

[3] Broadcom. 2018. 12Gb/s MegaRAID Tri-Mode Software. Retrieved from https://docs.broadcom.com/docs/MR-TM-SW-UG105.

[4] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the USENIX Conference on File and Storage Technologies.*

[5] Bianca Schroeder and Garth Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the USENIX Conference on File and Storage Technologies.*

[6] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, and Bianca Schroeder. 2020. A study of SSD reliability in large scale enterprise storage deployments. In *Proceedings of the USENIX Conference on File and Storage Technologies.*

[7] Jimmy Yang and Feng-Bin Sun. 1999. A comprehensive review of hard-disk drive reliability. In *Proceedings of the Annual Reliability and Maintainability Symposium.*

[8] Yu Cai, Erich F. Haratsch, Onur Mutlu, and Ken Mai. 2012. Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'12).*

[9] Yu Cai, Saugata Ghose, Erich F. Haratsch, Yixin Luo, and Onur Mutlu. 2017. Error characterization, mitigation, and recovery in flash-memory-based solid-state drives. *Proc. IEEE* 105, 9 (2017), 1666–1704.

[10] Myungsuk Kim, Youngsun Song, Myoungsoo Jung, and Jihong Kim. 2018. SARO: A state-aware reliability optimization technique for high density NAND flash memory. In *Proceedings of the Great Lakes Symposium on VLSI.*

[11] Micron. 2011. TN-29-59: Bad Block Management. Retrieved from https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2959_bbm_in_nand_flash.pdf.

[12] Samsung. 2014. Samsung V-NAND Technology, White Paper. Retrieved from https://studylib.net/doc/8282074/samsung-v-nand-technology.

[13] Jacob Alter, Ji Xue, Alma Dimnaku, and Evgenia Smirni. 2019. SSD failures in the field: Symptoms, causes, and prediction models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis.*

[14] Over-provisioning. 2020. Retrieved from https://www.seagate.com/tech-insights/ssd-over-provisioning-benefits-master-ti/.

[15] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv.* 26, 2 (1994), 145–185.

[16] Serial AT Attachment. Retrieved from https://sata-io.org/.
[17] NVM Express. Retrieved from https://nvmexpress.org/resources/specifications/.
[18] SCSI Storage Interfaces. Retrieved from http://www.t10.org.
[19] Seagate Technology. 2011. Reducing RAID Recovery Downtime. Retrieved from https://www.seagate.com/files/staticfiles/docs/pdf/whitepaper/tp620-1-1110us-reducing-raid-recovery.pdf.
[20] Mai Zheng, Joseph Tucek, Feng Qin, and Mark Lillibridge. 2013. Understanding the robustness of SSDs under power fault. In *Proceedings of the USENIX Conference on File and Storage Technologies*.
[21] Ying Y. Tai. 2016. High performance FTL for PCIe/NVMe SSDs. In *Proceedings of the Flash Memory Summit*.
[22] Shunzhuo Wang, Fei Wu, Chengmo Yang, Jiaona Zhou, Changsheng Xie, and Jiguang Wan. 2019. WAS: Wear aware superblock management for prolonging SSD lifetime. In *Proceedings of the Design Automation Conference*.
[23] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the Workshop on Hot Topics in Storage and File Systems*.
[24] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proceedings of the USENIX Conference on File and Storage Technologies*.
[25] Ulink. DriveMaster. 2019. Retrieved from https://ulinktech.com/products/drivemaster-8-enterprise-sas/.
[26] Jens Axboe. 2020. FIO. Retrieved from https://github.com/axboe/fio.
[27] Iometer. 2014. Retrieved from http://www.iometer.org/.
[28] Eden Kim. 2014. Enterprise Applications: How to Create a Synthetic Workload Test. Retrieved from https://www.snia.org/sites/default/files/EdenKim_Enterprise_Applications_WorkLoad_Test_SDC_2014.pdf.
[29] Youngjae Kim, Sarp Oral, Galen M. Shipman, Junghee Lee, David A. Dillow, and Feiyi Wang. 2011. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *Proceedings of the Symposium on Mass Storage Systems and Technologies*.
[30] Ulrich Hansen. 2012. The SSD Endurance Race: Who's Got the Write Stuff? Retrieved from https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TC11_Hansen.pdf.
[31] Richard R. Muntz and John C. S. Lui. 1990. Performance analysis of disk arrays under failure. In *Proceedings of the International Conference on Very Large Databases*.
[32] Mark Holland and Garth Gibson. 1992. Parity declustering for continuous operation in redundant disk arrays. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems*.
[33] G. A. Alverez, Walter A. Burkhard, L. L. Stockmeyer, and Flaviu Cristian. 1998. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the International Symposium on Computer Architecture*.
[34] Siu-Cheung Chau and Ada Wai-Chee Fu. 2002. A gracefully degradable declustered RAID architecture. *Clust. Comput.* 5, 1 (2002), 97–105.
[35] Jiguang Wan, Jibin Wang, Changsheng Xie, and Qing Yang. 2013. S2-RAID: Parallel RAID architecture for fast data recovery. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2013), 1638–1647.
[36] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 2018. RAID+: Deterministic and balanced data distribution for large disk enclosures. In *Proceedings of the USENIX Conference on File and Storage Technologies*.
[37] Scott Shadley. 2011. SSD RAIN. Retrieved from https://www.micron.com/~/media/documents/products/technical-marketing-brief/brief_ssd_rain.pdf.
[38] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. 2009. FRA: A flash-aware redundancy array of flash storage devices. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*.
[39] Soojun Im and Dongkun Shin. 2011. Flash-aware RAID techniques for dependable and high-performance flash memory SSD. *IEEE Trans. Comput.* 60, 1 (2011), 80–92.
[40] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. 2011. A lifespan-aware reliability scheme for RAID-based flash storage. In *Proceedings of the ACM Symposium on Applied Computing*.
[41] Yi Qin, Dan Feng, Jingning Liu, Wei Tong, Yang Hu, and Zhiming Zhu. 2012. A parity scheme to enhance reliability for SSDs. In *Proceedings of the International Conference on Networking, Architecture, and Storage*.
[42] Heejin Park, Jaeho Kim, Jongmoo Choi, Donghee Lee, and Sam H. Noh. 2015. Incremental redundancy to reduce data retention errors in flash-based SSDs. In *Proceedings of the International Conferece on Massive Storage Systems and Technology*.
[43] Jaeho Kim, Eunjae Lee, Jongmoo Choi, Donghee Lee, and Sam H Noh. 2016. Chip-level raid with flexible stripe size and parity placement for enhanced ssd reliability. *IEEE Trans. Comput.* 65, 4 (2016), 1116–1130.
[44] Bryan S Kim, Jongmoo Choi, and Sang Lyul Min. 2019. Design tradeoffs for SSD reliability. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

Received August 2020; revised December 2020; accepted February 2021