A Mirroring-Assisted Channel-RAID5 SSD for Mobile Applications

WEN PAN and TAO XIE, San Diego State University

Simply applying an existing redundant array of independent disks (RAID) technique to enhance data reliability within a single solid-state drive for safety-critical mobile applications significantly degrades performance. In this article, we first propose a new RAID5 architecture called channel-RAID5 with mirroring (CR5M) to alleviate the performance degradation problem. Next, an associated data reconstruction strategy called mirroring-assisted channel-level reconstruction (MCR) is developed to further shrink the window of vulnerability. Experimental results demonstrate that compared with channel-RAID5 (CR5), CR5M improves performance up to 40.2%. Compared with disk-oriented reconstruction, a traditional data reconstruction scheme, MCR on average improves data recovery speed by 7.5% while delivering a similar performance during reconstruction.

CCS Concepts: • **Computer systems organization → Reliability**; **Redundancy**;

Additional Key Words and Phrases: NAND flash, SSD, RAID, data reconstruction

ACM Reference format:
Wen Pan and Tao Xie. 2018. A Mirroring-Assisted Channel-RAID5 SSD for Mobile Applications. *ACM Trans. Embed. Comput. Syst.* 17, 4, Article 75 (July 2018), 27 pages.
https://doi.org/10.1145/3209625

1 INTRODUCTION

Safety-critical mobile applications are mobile applications whose failures could result in loss of life, significant damage to property or the environment, or serious detriment to national security. Typical examples of such applications include remote robotic surgery [32], aircraft flight control [16], wireless healthcare [35], and mobile emergency datacenter for disaster recovery [38]. These applications normally demand a high-performance, highly reliable, and energy-efficient storage system. Take remote robotic surgery, for example. Since in around two-thirds of fatal battlefield injuries death comes within 30min, medical response time in the battlefield is life-critical [32]. Remote robotic surgery would allow a military doctor, stationed in safe distance from the front line, to perform operations through a remote-controlled robot that carries out tasks such as cutting and dissecting for a wounded soldier [32]. During a surgery, high-resolution images taken from a wounded soldier by remote video cameras must be reliably stored into and then quickly accessed from a storage system. Obviously, data corruption or loss is prohibited as it could noticeably affect the quality of surgery. Also, the storage system needs to be energy-efficient as the entire system only has a limited power supply provided by a gasoline generator [32]. Another example

This work is sponsored by the U.S. National Science Foundation under Grant No. CNS-1320738.
Authors' addresses: W. Pan, San Diego State University, 5500 Campanile Dr, San Diego, CA, 92182; email: wpan@mail.sdsu.edu; T. Xie, San Diego State University, 5500 Campanile Dr, San Diego, CA, 92182; email: txie@sdsu.edu.

https://doi.org/10.1145/3209625

is wireless healthcare, which is the practice of medicine and public health supported by a number of wireless technologies and mobile devices [35]. It utilizes mobile communication devices (e.g., mobile phones) for collecting community and clinical health data, delivery of healthcare information to patients, real-time monitoring of patient vital signs, and direct provision of care [35]. Thus, a reliable storage system is essential for a wireless healthcare system where life-critical personal health data is collected, stored, and analyzed on a daily basis.

Since NAND flash memory–based solid-state drive (SSD) possesses some features such as high shock/temperature resistance and low energy consumption desirable for a portable environment, it becomes an increasingly popular storage device for safety-critical mobile applications. Besides, it can deliver a higher throughput than a hard-disk drive (HDD). Unfortunately, the reliability of flash memory is declining [6]. This is mainly because manufacturers are aggressively pushing flash memory into smaller geometries and each memory cell has to store more bits to increase capacity and lower cost [6]. As a result, the rate of increasing bit errors might exceed the capacity of ECC schemes [20], which leads to the occurrences of uncorrectable errors. In fact, a recent large-scale flash reliability field study [34] covering many millions of drive days in Google's data centers disclosed that comparing with traditional HDDs, SSDs have a higher rate of uncorrectable errors. The study showed that 26% to 60% of SSDs at least experienced one uncorrectable error during their first four years in the field. Also, it found that around 2% to 7% of SSDs developed failed chips (i.e., a chip with more than 5% bad blocks) during the first four years of their life [34]. Similarly, to understand failure properties and trends of SSDs, another research group analyzed data collected across a majority of SSDs at Facebook data centers over nearly four years and many millions of operational hours [25]. They discovered that the yearly rates of uncorrectable errors on the SSDs range from 15,128 for Platform D to 978,806 for Platform B with each platform having two SSDs [25]. Considering that a mobile environment is normally far more severe than a data center, it is safe to conjecture that the uncorretable error rate of an SSD in a mobile application could be even higher.

A natural solution to the data loss problem caused by uncorrectable errors is to employ a higher level data redundancy mechanism [34]. With the mechanism in place, when an SSD is not able to deliver a piece of data that it has previously stored, a data reconstruction process is launched to restore the data based on its associated redundant data [9]. One obvious way to provide a higher level data redundancy is to build an SSD array using an amended version of an existing redundant array of independent disks (RAID) technique (e.g., RAID4 or RAID5), which was originally designed for an HDD array [5, 12, 19, 29, 30]. However, although an SSD array might be appropriate for a mobile emergency datacenter built on a 25-foot self-contained truck [38], it is impractical for most safety-critical mobile applications like wireless healthcare [35], where the space and energy budget for a storage system are very limited. In these scenarios, a data redundancy mechanism within a single SSD is greatly needed [14, 18, 22, 42]. Fortunately, the internal multi-level parallelism and independent failure modes of individual flash chips provide us with a unique opportunity to apply a RAID-like data redundancy mechanism within a single SSD [10].

However, simply transplanting an existing RAID technique into the internal structure of an SSD causes some problems [42]. First, a study found that a channel-level RAID5 architecture significantly degrades performance when it serves small random write/update-dominant workloads [42]. The reason behind this is that for each update request, RAID5 has to perform extra read and write operations to renew its associated parity data [42]. Second, another investigation on a chip-level RAID5 structure unearthed that the extra write operations lead to more erases, which quickly consume the program/erase (P/E) cycles of an SSD [14]. Consequently, the reliability level and lifetime of an SSD were both reduced [14]. Finally, an unrecoverable data loss could occur when

a RAID5-structured SSD is undergoing a data reconstruction process. When a flash chip fails, a data reconstruction process is immediately launched to recover its data onto a new replacement chip. The time length of a data reconstruction process is also called *window of vulnerability*, for a subsequent chip failure within the time window can lead to an unrecoverable data loss [44]. Since RAID5 evenly distributes user write requests and internal parity-updating requests across all the chips, chips wear out at a similar rate. There is a possibility that a second chip could fail before an on-going data reconstruction process is over [25]. Unfortunately, a data reconstruction process of a traditional RAID5 architecture is usually lengthy. For example, the time to rebuild a single HDD can be on the order of several hours or even longer [33]. A larger size of window of vulnerability implies a higher probability of an unrecoverable data loss.

To alleviate the performance and endurance degradation problems as well as to minimize the window of vulnerability to enhance data reliability, in this article we first propose a mirroring-assisted channel-level RAID5 architecture called channel-RAID5 with mirroring (CR5M) for a single SSD employed in a safety-critical mobile application. Next, a new data reconstruction strategy called *m*irroring-assisted *c*hannel-level *r*econstruction (MCR) is developed to further accerlate data recovery speed. In a CR5M architecture, each channel is treated as an independent disk and it may have one or multiple chips attached. Multiple channels of an SSD are organized in a RAID5-like manner. In addition, a user-transparent mirroring chip is attached to each channel (see Figure 3). When a small write/update request arrives, it is dispatched to its destination channel. And then, it is concurrently written onto a particular data chip and the mirroring chip without updating its associated parity data. Clearly, the newly written data is protected by mirroring, whereas other data blocks in the same stripe are still protected by the present parity. The benefit of using a mirroring chip per channel is threefold: First, the overall performance can be boosted by postponing parity updating caused by small random writes/updates. Our experimental results show that compared with CR5 (channel-RAID5), a standard channel-level RAID5 organization, CR5M can improve performance by up to 40.3%. Second, the wear-out problem can be mitigated by postponing parity updating, which decreases the number of parity writes. Last but not least, data on the mirroring chips can help speedup data reconstrction through two aspects: These data do not need to be reconstructed and they can still serve partial user requests during data reconstruction. To prevent a mirroring chip from becoming a new performance and reliability bottleneck, a mirroring chip protection mechanism is proposed to limit its number of writes. To shrink the window of vulnerability, MCR utilizes the mirroring data to accelerate a data reconstruction process. Although the idea of MCR is straightforward, our experimental results demonstrate that it is effective. To evaluate the performance of CR5M and its associated data reconstruction strategy MCR, we largely extend a validated SSD simulator called SSDsim [10], so that it can support CR5, CR5M, disk-oriented reconstruction (DOR) [8], and MCR. CR5M and MCR collaborate with a page-mapping FTL provided by SSDsim [10]. Experimental results from real-world traces show that CR5M outperforms CR5 on average by 19.8% in terms of mean response time. Also, compared with DOR on average MCR improves data recovery speed by 7.5% while delivering a similar performance during recovery. This article is a substantial extension of our preliminary study presented at Reference [42]. The rest of this article is organized as follows. In Section 2, we discuss the related work and motivation. The design and implementation details of CR5M and MCR are provided in Section 3. In Section 4, we evaluate the performance of CR5M and MCR. Section 5 concludes this article with a brief summary.

2 RELATED WORK AND MOTIVATION

In this section, we first introduce existing data redundancy techniques. Next, traditional data reconstruction schemes are briefly summarized. Finally, we provide the motivation of this research.

2.1 Existing RAID Techniques

Much effort was devoted decades ago to enhancing the performance of a standard RAID architecture (e.g., RAID-5) for HDD arrays. A representative work is log-structured arrays (LSA) [24], which is inspired by the log-structured file system. It leverages non-volatile cache to largely delay parity updating so that it is more efficient than a conventional RAID-5 organization. Since SSD is replacing HDD in different computing platforms ranging from laptops, personal computers, servers, to supercomputers, various new techniques dedicated for RAID-structured SSDs have been proposed recently. Generally, they targeted two different levels: an SSD array [29, 30, 41, 43] and a single SSD [5, 12, 14, 18, 19, 22, 42].

Similar to an HDD array, an SSD array can largely boost performance and data reliability. However, due to SSD's unique needs, extra care must be taken to improve data reliability and performance. Since flash memory write speed is much slower than its read speed and parity updatings require extra flash memory accesses, most existing RAID techiques for SSD arrays focus on reducing write and parity updating costs. For example, Park et al., Wu et al., and Pan et al. each developed a new SSD RAID architecture [29, 43]. Park et al. proposed a heterogeneous RAID4, which utilizes an HDD to serve as the parity disk for a RAID4 SSD array [30]. Wu et al. also used HDDs to improve the performance and reliability of an SSD array, but in a different way. Their LDM (log disk mirroring) scheme employs two HDDs, which are mirrored as a write buffer that temporally absorbs small write requests [43]. Focusing on a pure SSD array, Pan et al. developed the SPD-RAID4 (splitting parity disk-RAID4) technique where two small-capacity SSDs both serve as the parity disks [29]. We compared the performance of a preliminary version of CR5M with CR1 (channel-RAID1) and CR4 (channel-RAID4) in [42]. We found that CR4 always performs the worst among the three RAID formats (i.e., CR1, CR4, and CR5), because its dedicated parity channel becomes a performance bottleneck [42]. In addition, RAID4 is seldom employed in a real-world disk array. Therefore, we ignored CR4 in this article. As for CR1, we found that it always outperformed CR5 and CR5M, because it does not need to maintain any parity updates. However, CR1 is impractical as it has to shrink the user-visible capacity of a flash SSD by 50%. Thus, we concentrated on comparing CR5M with CR5 in this research.

Since a multi-SSD RAID array is impractical for many safety-critical mobile applications where only one SSD can be deployed, a single SSD with data redundancy protection is greatly needed. Fortunately, the internal structure of an SSD offers parallelism at multiple levels (e.g., channel-level and chip-level) [10], which makes building a RAID-like structure within an SSD possible. Current RAID techniques for a single SSD are mainly applied at either chip-level [5, 12, 14, 18] or channel-level [19, 22, 42]. Lee et al. proposed a new chip-level RAID scheme, which can dynamically change the size of striping groups to cope with the increasing bit error rates [18]. Kim et al. developed eSAP (elastic striping and anywhere parity)-RAID, a RAID scheme that allows flexible stripe sizes and parity placement [14]. Im and Shin used delayed parity updating and partial parity caching techniques to reduce the number of write operations for parity updatings [12]. Their parity caching technique also reduces the number of read operations caused by calculating a new parity. Instead of simply applying a RAID technique to a set of flash devices, Greenan et al. proposed a RAID4-like SSD architecture where FTL is moved from individual devices into the host [5]. Also, writes are managed in a log-structured manner to avoid read-modify-write. Similar to Reference [12], Reference [5] used an non-volatile RAM (NVRAM) to hold the parity temporarily to prevent frequent parity updatings for write requests. MacFadden et al., however, developed a channel-level RAID1 (i.e., mirroring) architecture called SIRF-1 (single internally redundant flash) where data are mirrored across SSD channels so that read performance and data reliability are both improved [22]. The FRA scheme also uses a delayed parity updating scheme [19]. However,

it does not rely on cache to store the parity data. Instead, the parity updates are sent into a parity generation queue. Only when there appears to be an idle period, FRA pops the first parity update in the parity generation queue. The parity is then calculated and written into the flash storage at idle time. To reap the benefits of data redundancy while avoiding its side-effects on performance and longevity, a preliminary version of CR5M was developed in Reference [42] without a data reconstruction strategy.

2.2 Existing Data Reconstruction Approaches

None of existing SSD RAID techniques [14, 15, 18, 22, 41, 42] attempted to address the data reconstruction problem in case of a device (e.g., an SSD or a chip of an SSD) failure. After all, one of the major purposes of a RAID technique is to quickly recover the data of a failed device onto a replacement device [8]. To the best of our knowledge, the only work on SSD data reconstruction reported in the literature is Reference [21]. Thus, existing SSD RAID techniques still largely rely on traditional HDD-oriented data reconstruction strategies, which are summarized below.

Generally, traditional data reconstruction strategies can be divided into two categories. Strategies in the first category mainly focus on optimizing the workflow of reconstruction [8, 9, 39, 44]. Typical examples are stripe-oriented reconstruction (SOR) and disk-oriented reconstruction (DOR) [8, 9]. SOR uses low-priority requests to rebuild data to minimize the impact of reconstruction on user response times, whereas DOR aggressively absorbs all spare bandwidth for data reconstruction [8, 9]. The basic difference between the two strategies is that SOR creates a set of reconstruction processes associated with stripes while DOR generates a group of processes with each corresponding to one disk. Compared with SOR [9], DOR always results in a shorter data reconstruction time. Since shrinking data reconstruction time is the highest priority for a safety-critical mobile application, we choose DOR as a baseline algorithm in our evaluation of MCR. Popularity-based Reconstruction Optimization (PRO) [39] and MultIlevel Caching-based Reconstruction Optimization (MICRO) [44] are two reconstruction optimization schemes. They are not a stand-alone reconstruction approach like DOR. However, they can be incorporated into most existing reconstruction approaches to boost their performance. Both PRO and MICRO have been integrated into DOR [39, 44]. PRO is an algorithm to optimize an existing reconstruction approach by rebuilding highly popular data units of a failed disk prior to rebuilding other units [39]. However, MICRO [44] collaboratively utilizes storage cache and disk array controller cache to diminish the number of physical disk accesses caused by data reconstruction.

Data layouts like RAID5 significantly degrade performance due to a 100 percent workload increase for each surviving disk during reconstruction [8]. To solve this problem, data reconstruction algorithms in the second category concentrate on optimizing the data layout of an HDD array. For example, Muntz and Lui suggested a data placement scheme called declustering, which substantially improves performance [27]. Along the same line, Alvarez et al. further developed two data layout algorithms, PRIME and PELPR [1]. Wan et al. proposed s^2-RAID, which divides each disk into multiple logic disk partitions so that data reconstruction can be done in parallel [40]. Liu et al. proposed PDB (popular data backup), a data reconstruction algorithm for a RAID4 SSD array.

2.3 Motivation

The reliability of flash memory is declining due to aggressive scaling of the device dimension and the multilevel-cell technologies [6]. Thus, a data redundancy mechanism within a single SSD is greatly needed to prevent data loss, which is prohibited by a safe-critical mobile application. Based on our prior investigations [29, 42], we found that simply transplanting an existing data redundancy mechanism like RAID5 into the internal structure of an SSD does not work as it overlooks the unique features of flash memory such as wear-out and out-of-place updates. The great need

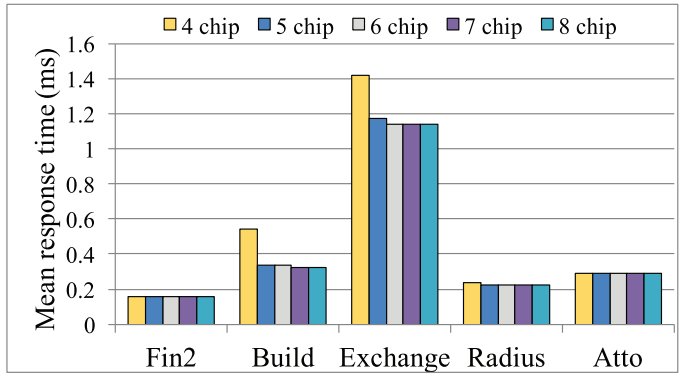


Fig. 1. The impact of the number of chips per channel on performance.

of a data redundancy mechanism suitable for a single SSD as well as the insights from our own studies [22, 29, 41, 42] and other researchers' work [3, 14, 15, 18, 30] motivate us to develop a new RAID technique that can not only improve performance and endurance but also facilitate data reconstruction.

There are four levels of parallelism in an SSD: channel-level, chip-level, die-level, and plane-level. Hu et al. discovered that the optimal priority order of parallelisms in an SSD should be the channel-level parallelism first and the chip-level parallelism last [10]. In other words, channel-level parallelism has the most significant impact on performance, whereas chip-level parallelism only minimally influences the performance of an SSD. There are two implications of their finding. First, a new RAID architecture like CR5M should be developed at the channel-level, which can offer the highest level of parallelism within an SSD. Second, reducing the number of chips per channel by one would not noticeably degrade SSD performance. To verify this implication, we conduct a group of experiments on an SSDsim-simulated 4-channel CR5-structured SSD. The experimental results are shown in Figure 1. The details of the simulator, the traces, and configurations can be found in Section 4. We change the number of chips per channel from four (i.e., 4 chip) to eight (i.e., 8 chip). Our experimental results demonstrate that after the number of chips per channel reaches four (e.g., Fin2, Radius, ATTO) or five (e.g., Build and Exchange) further increasing it does not improve performance. The finding from Reference [10] and our investigation results shown in Figure 1 inspire us to use one of chips per channel as a dedicated mirroring chip so that the number of parity updates can be diminished. Turning one chip per channel into a user-transparent mirroring chip does reduce user-visible SSD capacity and chip-level parallelism. Fortunately, its impact on performance is minor considering that modern SSDs usually have more than four chips per channel [13, 14] and their capacities are normally in the range of hundreds of GBs [26].

3 DESIGN AND IMPLEMENTATION

In this section, we first present design and implementation details of CR5M, a new channel-level RAID5 technique for a single SSD. Next, we describe how MCR, a data reconstruction strategy based on CR5M, is developed.

3.1 The CR5M Architecture

We first briefly introduce how a standard RAID5 mechanism updates parity data and why parity updating hurts the performance and endurance of an SSD. Next, we present the architecture of CR5M followed by its design and implementation details.

3.1.1 CR5 (Channel-Level RAID5) Basics. An SSD consists of two major components: an SSD controller and a flash memory part. The SSD controller manages the flash memory through a software layer called FTL (flash translation layer), which provides an interface to a host computer. The main functions of an FTL include address mapping, wear-leveling, and garbage collection [15]. The flash memory part is organized as a multi-level hierarchical structure, which has several channels (see Figure 3). Multiple chips are attached to each channel and each chip consists of a couple of dies. A die is composed of several planes with each of them having thousands of blocks. Typically, a block has 64 or 128 pages and the size of each page ranges from 2 to 16KB [42]. While a read/write operation executes on a page granularity, an erase operation can only be carried out on a block. The parallelisms of an SSD can be classified into four levels: channel-level, chip-level, die-level, and plane-level [10]. Although in theory a RAID-like architecture can be built at any of these four levels, existing RAID techniques only target either chip-level [14, 18] or channel-level [22, 42]. Since a flash memory chip is the smallest replaceable unit, Lee et al. and Kim et al. proposed RAID techniques at the chip-level [14, 18]. However, we decided to select the channel-level, because a channel can act more like an independent disk, which makes building an array easier. Besides, channel-level parallelism can provide the most concurrency [10]. In fact, an investigation on SSD internal multi-level parallelism discovers that the channel-level parallelism should be given the first priority as increasing the number of channels results in the largest performance improvements [10]. Modern SSDs normally have at least four channels with each having multiple chips [17, 26, 31]. For example, a Micron M500IT mSATA SSD has eight channels and each channel has five chips [26].

The proposed CR5M architecture is essentially an enhanced version of a standard RAID5 structure, which is one of the most widely used RAID formats in real production. Before we present the design of CR5M, we first briefly introduce how a standard RAID5 mechanism updates parity data and why parity updating hurts the performance and endurance of an SSD. Most RAID formats (e.g., RAID5) are designed to provide increased performance by distributing (or striping) data across a set of physical drives in a disk array. Each X bytes of data, which is called a strip, is placed on a different drive in the array on a rotating basis. A stripe is one complete row of strips across all physical drives in an array. In a channel-level RAID5 (i.e., CR5) SSD with n channels, a stripe consists of n-1 data strips and 1 parity strip. The n strips are evenly distributed across the n channels. The size of a strip (i.e., strip size) is normally set equal to the size of one flash memory page for convenience. The stripe width is the number of data strips in a stripe. The stripe size is equal to the product of strip size and stripe width. For example, a stripe shown in Figure 2(a) consists of five data strips (i.e., *D1*, *D2*, *D3*, *D4*, *D5*) and one parity strip (i.e., *P1*). The six strips are evenly distributed across the six channels and each channel is treated as an independent disk. The value of *P1* is equal to $D1 \oplus D2 \oplus D3 \oplus D4 \oplus D5$. User write requests can be divided into two groups based on their sizes: full-stripe writes and partial-stripe writes. A full-stripe write is going to update all data strips within a stripe, whereas a partial-stripe write only modifies part of the data strips in a stripe. Obviously, a full-stripe write incurs the least overhead of parity updating as the new parity strip can be directly computed without the need of any pre-reading. In contrast, a partial-stripe write like the one shown in Figure 2(a) only modifies part of the data strips in a stripe. In this case, a new parity strip needs to be calculated, which requires a number of pre-reading operations. There are two common methods to do so: RMW (read-modify-write) and RCW (read-construct-write). Now assume that a three-page write request is going to update the contents of *D1*, *D2*, and *D3* (see Figure 2(a)). While Figure 2(a) shows how RMW updates the parity strip *P1*, Figure 2(b) illustrates how RCW accomplishes the same task. In this example, RCW is more favorable as it incurs a fewer number of pre-readings. In fact, for each parity updating process, RAID5 always chooses

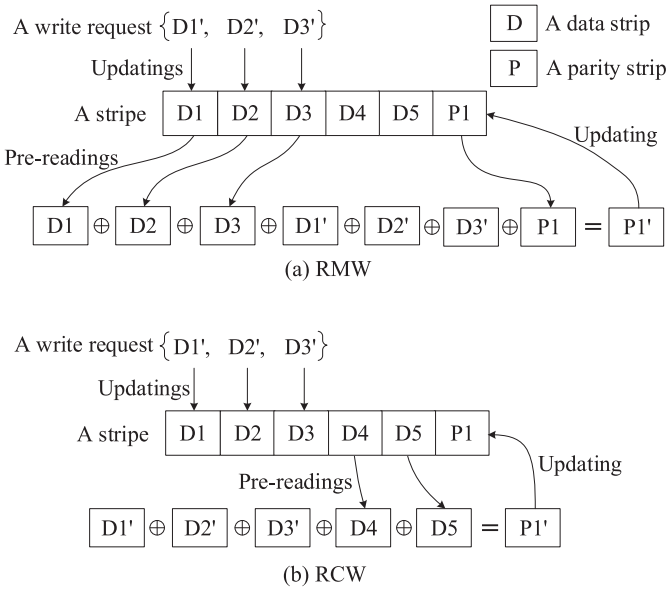


Fig. 2. Two parity updating methods in RAID5.

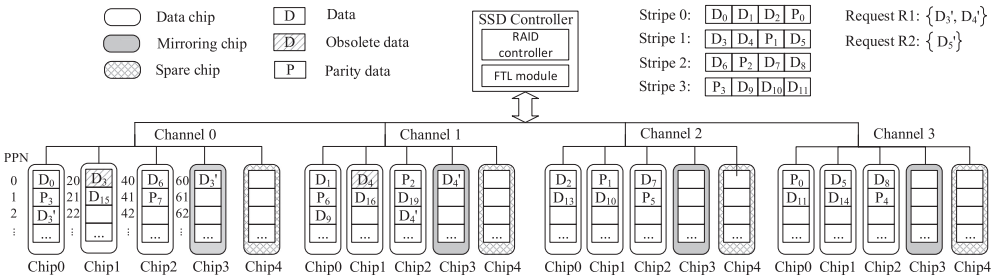


Fig. 3. The architecture of CR5M.

the method that can lead to a fewer number of pre-readings. If the numbers are equal, then RCW is adopted.

Clearly, a parity updating process degrades performance as it incurs extra read (i.e., pre-reading) and write (i.e., new parity write back) operations, which substantially increase the response times of user requests. The experimental results from Reference [42] demonstrate that compared with a 4-channel SSD without any RAID mechanism the mean response time of a CR5-structured SSD increases from 25% to 160% on various real-world workloads. Moreover, since a write request would cause at least one parity updating, a CR5-based SSD wears out quickly when the majority writes are small random updates, which hurts its endurance.

3.1.2 The Architecture of CR5M. To alleviate the performance and endurance degradation problems, we propose a new RAID mechanism called CR5M, which is a channel-level RAID5 structure with mirroring [42]. The architecture of CR5M for an SSD with four channels is shown in Figure 3. In the SSD, five chips are attached to each channel: three data chips, one mirroring chip, and one spare chip (i.e., replacement chip in MCR). The data placement of CR5M is similar to that of a

standard RAID5 format. For example, stripe 0 includes four strips: $D0$ (chip 0 on channel 0), $D1$ (chip 0 on channel 1), $D2$ (chip 0 on channel 2), and $P0$ (chip 0 on channel 3) (see Figure 3). In CR5M, the size of each strip is equal to the size of one flash page. The key feature of CR5M is that a mirroring chip is introduced to each channel. Also, CR5M assumes that each channel has at least one spare chip, which can be used as a replacement chip once a working chip (i.e., a data chip or a mirroring chip) stops to function. The spare chips do not serve any requests when an SSD is in the normal mode. They are reserved for the purpose of data reconstruction in case that one working chip on a channel fails. Like a traditional RAID5 technique, CR5M can only tolerate one chip failure at a time. Moreover, each channel only allows at most one chip failure if there is only one spare chip per channel. Both the mirroring chips and spare chips are transparent to users, and thus, they do not contribute to the user-visible total capacity of an SSD. The assumption of mirroring and spare chips is reasonable due to two facts. First, the price of flash memory is declining quickly [23], and thus, trading a few extra spare chips for an enhanced level of reliability is justifiable for an SSD used in a safety-critical mobile application. For example, by March 2016 the price of a 250GB SSD runs between \$52 to \$81, which leads to an average per-gigabyte cost of \$0.27 [23]. Second, CR5M turns one data chip per channel into a mirroring chip, which reduces the number of user-visible data chips per channel by one. However, using one fewer data chips per channel only has trivial impact on the performance of an SSD as shown in Figure 1. Although CR5M also uses mirroring mechanism in a RAID5 organization, it is quite different from a traditional HDD-oriented RAID51 format. In a RAID51 HDD array, there are two identical RAID5 sub-systems so that each data has two copies. In CR5M, only part of data has an identical copy on a mirroring chip and there is only one RAID5 system.

In addition to a normal RAID5-style write function, CR5M has a unique mirroring write (MW) function to utilize the mirroring chips to better serve a partial-stripe write request. Unlike a normal write, an MW operation writes a page of data into its destination data chip and the corresponding mirroring chip simultaneously. Since chips on the same channel can work in an interleaving way, the overhead of an MW operation is close to that of a write without a parity updating. Note that an MW operation is prohibited as long as one of the involved mirroring chips is busy. The reason is that waiting for a busy mirroring chip could cancel the benefits of an MW operation. To make sure that a mirroring chip does not wear out quicker than a data chip, CR5M uses a mechanism to confine the number of writes on it (see Section 3.1.3). In what follows, we use an example to illustrate how an MW operation works.

Figure 3 provides an example of how a partial-stripe write can be carried out by MW operations. The stripe 1 shown in Figure 3 consists of three data strips D_3, D_4, D_5, and a parity strip P_1. Assume that a partial-stripe write request *R1* is going to update D_3 and D_4. The conventional RAID5 would write D'_3 (i.e., a new version of D_3) and D'_4 into a data chip on channel 0 and a data chip on channel 1, respectively. And then it would mark D_3 and D_4 as invalid. Next, it would generate a pre-reading request on D_5 and then calculate the new parity P'_1, which is equal to $D'_3 \oplus D'_4 \oplus D_5$. Finally, it would write P'_1 back on a data chip on channel 2. However, CR5M only needs two concurrent MW operations to serve *R1* if the mirroring chip of channel 0 and the mirroring chip of channel 1 are both idle. While one MW operation writes D'_3 to the chip 0 of channel 0 and the mirroring chip of channel 0 simultaneously, the other MW operation writes D'_4 to the chip 2 of channel 1 and the mirroring chip of channel 1 at the same time. Note that D'_3 and D_3 are required to be on the same channel but not necessarily on the same chip. In fact, D'_3 can be written onto any idle data chip on channel 0, which speeds up an MW operation. Please refer to Section 3.1.3 for the details of a CR5M channel/chip selection scheme. The obsolete D_3 and D_4 are still valid now and will be invalidated later. The reason is that they can still help recover D_5 in case that it is corrupted. The two MW operations do not require any pre-reading, new parity calculating, and new parity

updating operations. And thus, CR5M serves the request *R1* more rapidly than a traditional RAID5 does. In the case that a subsequent request is going to read D'_3, either chip 0 on channel 0 or the mirroring chip on channel 0 can serve it.

Now assume that after a while another partial-stripe write request *R*2 arrives and it is going to update D_5 in stripe 1 (see Figure 3). At this moment, since all data strips in stripe 1 will be changed, the parity strip *P*1 as well as the obsolete data strips *D*3 and *D*4 are all no longer needed. CR5M first writes *D*5′ onto a data chip on channel 3. And then it pre-reads *D*3′ and *D*4′ from channel 0 and channel 1, respectively. Next, it calculates a new parity *P*1′, which is equal to $D3' \oplus D4' \oplus D5'$. The new parity *P*1′ will then be written onto any data chip on channel 2. Finally, CR5M invalidates *D*3, *D*4, *D*5, *P*1, *D*3′ on the mirroring chip of channel 0, and *D*4′ on the mirroring chip of channel 1. All these invalidated strips (i.e., pages) will be eventually reclaimed by a garbage collection (GC) mechanism in the FTL of an SSD. At this time, the new stripe 1 (i.e., *D*3′, *D*4′, *D*5′, *P*1′) returns to a standard RAID5 format, which has no data protection from a mirroring chip. The process of serving request *R*2 is not shown in Figure 3 to make the figure clear. If request *R*2 is going to update *D*4′ rather than *D*5, then CR5M will simply write *D*4′′ onto any data chip on channel 1 and the mirroring chip. Next, it will invalidate *D*4′ on both chips 2 and 3 of channel 1 as they are no longer needed. For each mirrored data strip, CR5M only maintains its latest version.

CR5M enhances performance and reliability from the following aspects. First, the mean response times of many partial-stripe writes are reduced as additional pre-reading and new parity calculating operations are eliminated. This is because these writes are served by the MW function instead of a traditional RAID5 write function. The statistics from our experiments show that 35% to 91% partial-stripe writes can be processed by the MW function. In the case that all data strips of a stripe are eventually updated, some pre-reading operations are still needed to calculate a new parity strip. However, the parity updating process has been largely postponed, which improves performance. This is because the number of parity calculations are reduced as only the latest version of a frequently updated stripe is needed for the new parity calculation. Second, an MW operation stores two copies of a data strip, which leads to an improved read performance, because a subsequent read request can still be served from the mirroring chip if the data chip is busy, or vice versa. Third, the data of a failed data chip can be quickly recovered, because part of its data have a copy on the mirroring chip, and thus, do not need to be reconstructed. A quick data recovery enhances data reliability.

3.1.3 The Design and Implementation of CR5M. In this section, we first introduce an extended address mapping table, which is the most important data structure of CR5M. Next, we explain how CR5M processes each read/write request. Finally, an algorithm of CR5M write request processing is provided.

In both normal mode and degraded mode, the RAID controller (see Figure 3) has to decide how to handle a request. A traditional mapping table entry consists of two fields: logical page number (LPN) and physical page number (PPN), to record the logical and physical address of a flash page. Since the MW function of CR5M sometimes needs to store a copy of a data strip on a mirroring chip, an extended mapping table is needed. In each entry of the extended mapping table, an additional field called mirroring address (MA) is used to indicate whether there exists a mirrored data of the current entry. If the value of the field is NULL, then the entry does not have a mirrored data. Otherwise, an address will be stored in the MA field, which points to an entry of a mirroring table. Each entry of a mirroring table has two fields: expired physical page number (EPPN) and mirroring physical page number (MPPN). While EPPN records the physical page address of an obsolete data, MPPN logs the physical address of a mirrored data. A physical address (e.g., PPN,

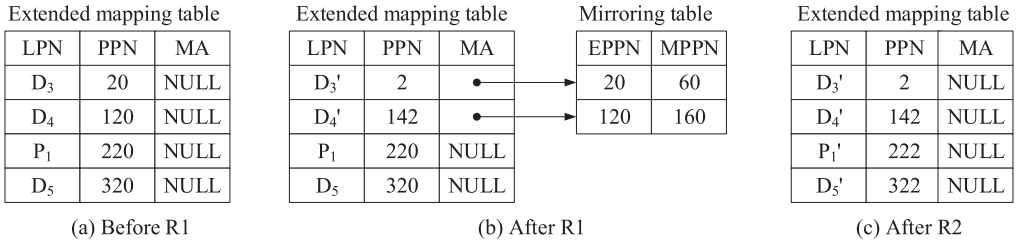


Fig. 4. An example of the extended mapping table.

MPPN, and EPPN) is a concatenation of channel number, chip number, die number, plane number, block number, and page number.

CR5M utilizes a semi-dynamic allocation scheme. The channel number is decided statically by the least significant bits of a logical address so that sequential pages can be distributed onto different channels to form a RAID5 organization. For each one-page write request, CR5M dispatches it to a particular channel based on its LPN using the following equation: the destination channel of a request = the request's LPN% the number of channels in an SSD. Thus, at the channel-level CR5M almost evenly distributes writes across all channels. After a one-page write request is sent to a particular channel, a classical wear-leveling method can start to work with CR5M to achieve wear leveling. In particular, it can now decide on which data chip, which die, which plane, and which block the request will be sent to so that writes will be evenly distributed among all blocks under the channel. In what follows, we use two request processing examples (i.e., $R1$ and $R2$) shown in Figure 3 to explain how these two tables are utilized. For illustration purpose, we assume that each chip shown in Figure 3 only has 20 pages. Figure 4 provides three snapshots of the two tables, which only cover stripe 1 shown in Figure 3. Before the arrival of $R1$, the PPN of $D3$, $D4$, $P1$, $D5$ is 20, 120, 220, 320, respectively (see Figure 4(a)). After $R1$ is processed, $D3'$ and $D4'$ have been written onto a data chip and the corresponding mirroring chip, respectively. Now the contents of the two tables are shown in Figure 4(b). After $R2$ is served, stripe 1 returns to a standard RAID5 format (see Figure 4(c)). We assume that $P1'$ and $D5'$ have been written to PPN 222 on channel 2 and PPN 322 on channel 3, respectively. Now, we show the sizes of the two tables through an example. Assume that the size of each flash page is 4KB and each address (i.e., MA, PPN, MPPN, or EPPN) is a 32-bit binary number. So, the maximal possible capacity of a CR5M-based SSD is 16TB (i.e., $2^{32} * 4KB = 16TB$). Since a LPN is only used as an index to locate a PPN, it does not need to be stored in the extended mapping table shown in Figure 4. Thus, each entry in the extended mapping table takes 64 bits. Similarly, each entry in the mirroring table requires 64 bits (see Figure 4). Assume that a CR5M-based SSD has four channels and each channel has three data chips, one mirroring chip, and one spare chip, as shown in Figure 3. Assume that each chip (either a data chip or a mirroring chip) has 2^{20} 4KB pages. Thus, the capacity of each chip is 4GB. The overall capacity of the CR5M-based SSD is then 64GB (i.e., 4 channels $*4$ chips $*4GB = 64GB$). Note that the number of entries in the extended mapping table is only decided by the aggregate capacity of all data chips. Similarly, the number of entries in the mirroring table is only determined by the aggregate capacity of all mirroring chips. Therefore, the number of entries in the extended mapping table is equal to the total number of pages of all data chips, which can be calculated by $2^{20} * 4 * 3 = 12M$. So, the size of the extended mapping table is $12M * 64$ bits = 96MB. Similarly, the size of the mirroring table is 32MB. In a nutshell, for a 64GB CR5M-based SSD, the total size of its two mapping tables is 128MB, which is about 0.2% of its capacity. A modern flash SSD product normally has an on-chip DRAM module whose size ranges from hundreds of MB to even 1GB. For example, a 120GB Samsung 840 EVO Series Solid State Drives (model # MZ-7TE120BW) has a 256MB on-chip DRAM as a

cache [28]. In addition, the price of an on-device DRAM is fairly stable at a very low level. Micron's average DRAM selling price per gigabit was as low as one dollar [37]. Furthermore, the price of DRAM will continue to decline with the advances of manufacturing technologies [37]. Therefore, we argue that caching 128MB mapping tables in an on-flash DRAM becomes affordable for a modern 64GB flash SSD. In the case that the two tables are too large to be accommodated in an on-flash cache, we can store them in flash and only place popular mappings in cache like DFTL [7] does.

After a write request arrives, if its address range spans two or more adjacent stripes (e.g., stripe 0 and stripe 1 shown in Figure 3 are two adjacent stripes), the RAID controller first splits it into multiple sub-requests such that each of them is either a full-stripe request or a partial-stripe request. The sub-requests are served sequentially and each of them only accesses a particular stripe. Next, the RAID controller further divides each sub-request into multiple one-page requests, which will then be dispatched to different channels. CR5M's semi-dynamic allocation scheme decides which channel and chip a one-page request should be sent to. For each sub-request, the RAID controller needs to make a decision on which parity updating method to use: RMW, RCW, MW, or directly calculating parity without a pre-reading. Also, an MW operation concurrently writes a page of data onto two different chips. Thus, a subsequent read request needs to select one of them to retrieve the data.

Since one mirroring chip needs to store mirrored data from multiple data chips on the same channel, it could receive more writes than a data chip, and thus, wear out quicker as more writes cause more erasures. To make sure that a mirroring chip does not prematurely fail because of a high concentration of erasure cycles, CR5M employs a mechanism called *mirroring chip protection* to limit the number of MW operations on a mirroring chip so that it could not receive substantially more writes than a data chip. For each channel, CR5M keeps track of two variables: number of MW operations executed (i.e., $MWNo\,[ChannelNo]$) and average number of writes on a data chip (i.e., $AveWriteNo\,[ChannelNo]$). The two variables for each channel are initialized to zero. The value of $AveWriteNo\,[ChannelNo]$ is equal to the total number of writes on all data chips of channel *ChannelNo* divided by the number of data chips on that channel. A tunable parameter called *MWLimiter* is used to limit the number of MW operations on the mirroring chip of a channel. More precisely, for each channel, the value of $MWNo\,[ChannelNo]$ should be always no larger than the product of *MWLimiter* and $AveWriteNo\,[ChannelNo]$. Clearly, a larger value of *MWLimiter* leads to more performance improvement when an SSD is in the normal mode, because more MW operations can be carried out. However, there is one obvious drawback. When the mirroring chip on a channel receives more writes, it wears out quicker than a data chip. As a result, it could become a performance/endurance bottleneck of the SSD. In fact, an appropriate value of *MWLimiter* is workload-dependent. For a write-intensive trace, it is better to set *MWLimiter* larger than 1 as the mirroring chip needs to accept more writes than a data chip on the same channel does. In our experiments, *MWLimiter* is set to 1.1 so that the number of writes on a mirroring chip does not exceed 1.1 times of the average number of writes on a data chip on the same channel. This value is empirically found to be a good trade-off between performance and endurance for the five traces.

An algorithm of CR5M write request processing is shown in Algorithm 1. If a sub-request is a full-stripe write, then the RAID controller directly generates a new parity data by executing XOR operations on the request's data without any pre-reading. Next, it updates all data strips and the parity strip. Further, it invalidates all old strips. Finally, for each channel of an SSD, it adds one to the total number of writes on all data chips. If a sub-request is a partial-stripe write, then the RAID controller detects whether all involved mirroring chips are idle and for each involved channel whether its number of MW operations performed so far is no larger than the product of *MWLimiter* and the average writes on a data chip. If these two conditions hold, for each page of data, then an MW operation will be invoked to write it onto both a data chip and the responding

ALGORITHM 1: Processing a Write Request R

```
if R span n stripes and n ≥ 2 then
    Split R into n sub requests
else
    R itself is a sub-request;
end
for each each sub-request do
    if it is a full-stripe request then
        Directly calculate a new parity;
        Update the entire stripe including the parity strip;
        Invalidate all strips of the old stripe;
    else if MWNo ≤ MWLimiter* AveWriteNo for all involved channels and all involved mirroring chips
      are idle then
        Perform an MW for each one-page request;
        Update each request's entries in the two tables;
        ++ MWNo for each involved channel;
    else if RMW Pre-reading < RCW Pre-reading then
        Perform RMW to update data and parity strips
    else
        Perform RCW to update data and parity strips
    end
    for each involved channels do
        ++ total number of writes
        AveWriteNo = total number of writes/number of data chips on each channel
    end
end
```

mirroring chip simultaneously. And then, each data strip's multiple physical addresses (e.g., PPN, MPPN, and EPPN) are recorded in the extended mapping table and mirroring table. Next, for each involved channel, its number of MW operations executed is increased by one. If one of the two conditions does not hold, then CR5M chooses either RMW or RCW to update a stripe. If the number of pre-reading of RMW is fewer than that of RCW, then RMW will be selected to process data and parity updating. Otherwise, RCW is chosen. Still, for each involved channel, the total number of writes on all data chips is increased by one. The processing of a read request is relatively simple. For each data strip of a read request, RAID controller first allocates its corresponding entry in the extended mapping table based on its LPN. If the MA field is NULL, then the data is read out from where PPN points to. Otherwise, the data can be fetched from either the data chip or the mirroring chip. Figure 5 provides a flowchart of the CR5M request processing procedure.

3.2 The MCR Data Reconstruction Strategy

Once a working chip (i.e., a data chip or a mirroring chip) fails, a CR5M-based SSD immediately enters into the degraded mode. In this mode, the SSD continues to serve external user requests but at a lower level of service while a background data reconstruction process is launched to recover the data of the failed chip on a replacement chip. After the data reconstruction process finishes, the SSD returns to the normal mode. Although various data reconstruction algorithms [1, 8, 9, 27, 29, 39, 40, 44] have been proposed, they share two common goals: speedup data rebuilding process (i.e., shrinking the window of vulnerability) and alleviating performance degradation during reconstruction. Since the efficiency of a data reconstruction algorithm not only impacts an SSD's performance during the degraded mode but also affects its data reliability, an efficient

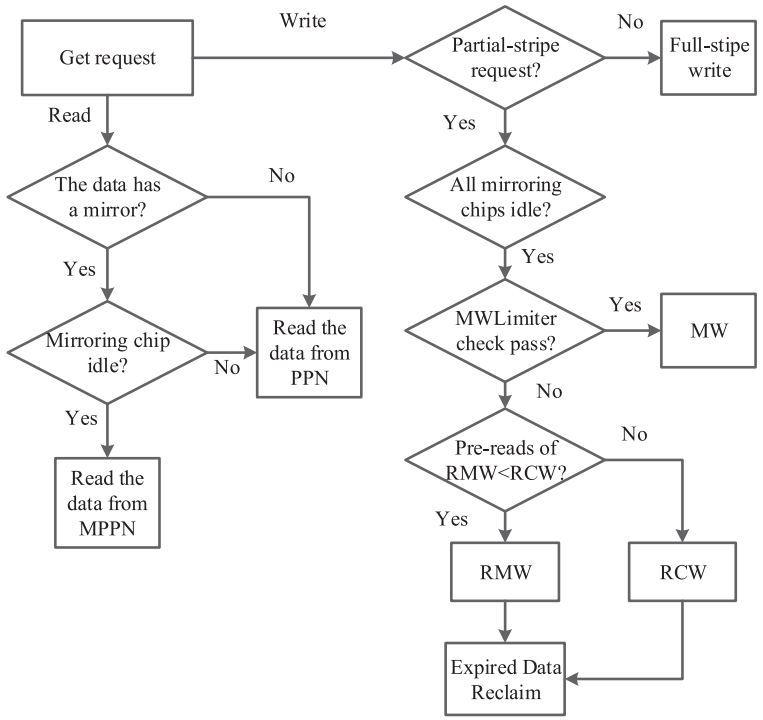


Fig. 5. The flowchart of CR5M request processing.

data reconstruction strategy is always a critical component of a parity-based RAID management system like CR5M. In this section, we present design and implementation details of MCR, a data reconstruction strategy developed for CR5M.

3.2.1 Recovering A Failed Mirroring Chip. Similar to CR5, CR5M can tolerate one working chip (i.e., either a mirroring chip or a data chip) failure at a time. We first demonstrate that all data on a failed mirroring chip can be recovered. Recall that the only way for a piece of data to be written onto a mirroring chip is through an MW operation (see Section 3.1.2). Besides, an MW operation guarantees that when a piece of data is written onto a mirroring chip, it is also simultaneously written onto a data chip on the same channel. As a result, each piece of data on a mirroring chip must have a replica on a data chip on the same channel, which makes recovering it simple. To recover all the data on a failed mirroring chip, MCR searches the extended mapping table. For each entry whose MA (i.e., mirroring address) field is not a NULL, it uses the address in the PPN field to locate the replica of a piece of data on the failed mirroring chip (see Figure 4(b)). And then, the data is copied to the replacement chip on the same channel. This process is repeated until all entries in the extended mapping table have been examined. After all data have been recovered on the replacement chip, the replacement chip now starts to serve as the mirroring chip so that the CR5M architecture is still maintained in the SSD. In fact, when recovering a failed mirroring chip, no data needs to be rebuilt. Rather, MCR simply copies the failed data from data chips to the replacement chip on the same channel (see Algorithm 2).

3.2.2 Recovering A Failed Data Chip. Apparently, when a CR5M-based SSD is in the normal mode, the mirroring chips help it postpone parity updating, and thus, its performance and

ALGORITHM 2: The MCR Data Reconstruction Strategy

```
switch failed chip do
    case mirroring chip do
        for each entry in extended mapping table with a non-NULL MA field do
            copy the data located in PPN to the replacement chip
        end
    case data chip do
        ———————Phase one———————
        for each each stripe do
            if one of its strips was on the failed data chip then
                switch stripe type do
                    case non-MW stripe do
                        recover the strip by a standard DOR
                    case sibling-MW stripe do
                        if failed data is user data then
                            recover the strip using obsolete data in DOR
                        else if failed data is parity data then
                            update parity and reclaim obsolete data
                        end
                    case MW stripe do
                        mark it as "MW"
                end
            end
        end
        ———————Phase Two———————
        for each stripe marked as "MW" do
            if the failed strip has been updated again so far then
                skip the stripe
            else
                copy its replica to the replacement chip
            end
        end
end
```

endurance are enhanced. After it falls into the degraded mode due to a data chip failure, the mirroring chip on the same channel can still provide support in two respects. First, part of data on the failed data chip have a replica on the mirroring chip. Therefore, these data do not need to be reconstructed. In fact, they can be simply copied from the mirroring chip to a replacement chip, which improves data recovery speed. Second, external user read requests visiting these data can still be served by the mirroring chip without the need to wait for the completion of their reconstructions. Thus, the mirroring chip can assist the SSD to reduce its performance degradation during the degraded mode.

Since the MCR data reconstruction strategy is derived from the classic DOR algorithm [8], we first briefly introduce how DOR reconstructs data. After one disk fails in a disk array with n disks, DOR launches two types of processes: n-1 data fetchers and one data deliverer. Each data fetcher process associates with one surviving disk. It keeps transferring relevant data or parity strips from its associated surviving disk to a central buffer. The data deliverer process grabs each piece of reconstructed data from the central buffer and then writes it onto a replacement disk. The central buffer consists of multiple pages and each page stores one strip. A page of the central buffer is

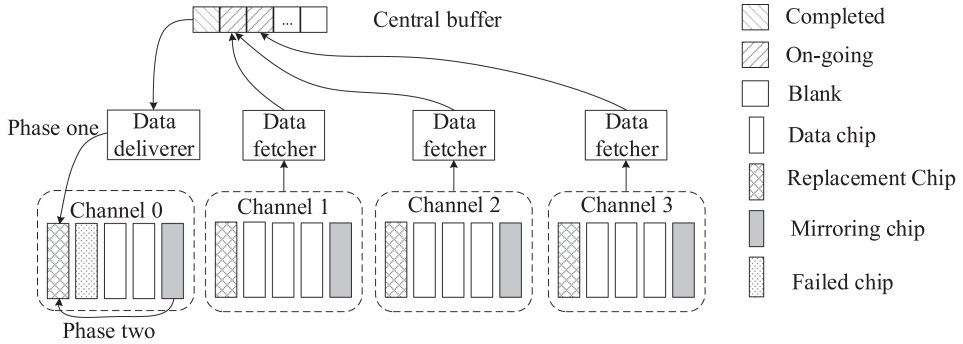


Fig. 6. The architecture of MCR.

assigned to a particular stripe when its reconstruction starts. After all surviving n-1 strips from the stripe are fetched and then sequentially XOR-ed with each other, the result (i.e., the once failed strip of the stripe) is written back to the buffer page. At this moment, the status of the buffer page changes from "Incomplete" to "Complete." The goal of DOR is to absorb all bandwidth that is not used by users. CR5M adopts a similar approach. Since CR5M treats each channel as an independent disk, MCR associates a data fetcher to a channel without any failed chip. A data deliverer keeps moving reconstructed pages from the buffer to the replacement chip on the channel that has a failed data chip. The size of a central buffer for MCR is set to eight pages in our experiments. In other words, MCR can reconstruct up to eight stripes simultaneously. We find that further increasing the size of a central buffer cannot noticeably improve the performance of data reconstruction. Figure 6 demonstrates how MCR works after the data chip 0 on channel 0 fails.

Stripes in a CR5M-based SSD can be classified into three categories: a non-MW stripe, a sibling-MW stripe, and an MW stripe. Based on stripe category and lost data type (data strip or parity strip), MCR uses different approaches to recover a lost strip. A data chip recovering process consists of two phases (see Figure 6). In phase one, MCR uses a DOR manner to recover all failed strips that belong to either a non-MW stripe or a sibling-MW stripe. In a non-MW stripe, none of its data strips has been updated by an MW operation. For example, if chip 0 on channel 0 shown in Figure 3 fails, stripe 0 is a non-MW stripe as none of its three data strips has ever been updated by an MW operation. Similarly, stripe 3 is also a non-MW stripe. MCR recovers *D0* and *P3* using a traditional DOR manner. Data strips in one stripe are called siblings. A stripe is called a sibling-MW stripe if it falls into one of the following two cases: (1) If the lost data is a data strip, then it has never been updated by an MW operation but at least one of its sibling data strips has been updated by an MW operation; (2) If the lost data is a parity strip, then at least one of data strips in the same stripe has been updated by an MW operation. For example, after the write request *R1* is processed and then chip 1 on channel 3 shown in Figure 3 fails, stripe 1 becomes a sibling-MW stripe, because although *D5* has not been updated by an MW operation, its sibling data strips *D3* and *D4* have been updated to $D3'$ and $D4'$ by an MW operation, respectively. In this case, MCR reconstructs *D5* by using its obsolete sibling data strips (i.e., *D3* and *D4*) and the out-of-date parity strip (i.e., *P1*) in a DOR manner. In the second case, the lost data is a parity strip (e.g., *P1*). For instance, assume that after the write request *R1* is processed and then chip 1 on channel 2 shown in Figure 3 fails, stripe 1 also becomes a sibling-MW stripe, because two of its data strips (i.e., *D3* and *D4*) have been updated by an MW operation. Instead of recovering the out-of-date *P1*, MCR updates it to $P1'$, which is equal to $D3' \oplus D4' \oplus D5$. Next, MCR reclaims the obsolete strips (i.e., *D3*, *D4*, *P1*).

After MCR has reconstructed all failed strips from non-MW and sibling-MW stripes, it enters into the second phase. In the second phase, MCR concentrates on recovering each data strip that

belongs to an MW stripe. In an MW stripe, the failed data strip has been updated by an MW operation. Note that a parity strip can never have a replica on the mirroring chip as an MW operation can only be performed on a data strip. Since each of these failed data strips has a replica on the mirroring chip, MCR simply copies these replicas onto the replacement chip to recover them. For example, assume that chip 0 on channel 0 fails right after the request $R1$ has been served and MCR is going to recover $D3'$ (see Figure 3). At this moment, stripe 1 is an MW stripe. Since a copy of $D3$ is on the mirroring chip (i.e., chip 3 on channel 0), MCR just copies $D3$ from the mirroring chip to the replacement chip. Obviously, phase two can be completed quickly, which makes MCR outperform DOR.

The algorithm of MCR is shown in Algorithm 2. MCR reconstructs failed data strips in MW stripes at last due to the following two considerations. First, these data strips have not been lost as their replicas still exist on the mirroring chip, which can serve relevant user read requests. Therefore, they are not in a hurry to be rebuilt. Second, since a failed data strip of an MW stripe is generally more popular than a data strip in other types of stripes, it is more likely that it has been updated once again after the start of a data reconstruction process. In that case, MCR no longer needs to reconstruct the failed data strip any more.

4 PERFORMANCE EVALUATION

In this section, we evaluate the performance of CR5M and MCR using both real-world traces and synthetic benchmarks on a simulator that is extended from SSDsim [10]. We first explain how we extend SSDsim [10], a hardware-validated single SSD simulator, so that it can support CR5M, CR5, MCR, and DOR. Next, we introduce experimental setup including simulator parameter configurations and the characteristics of the five real-world traces. Finally, we evaluate CR5M and MCR, respectively.

4.1 The Simulator

SSDsim is an event-driven, multi-tiered single SSD simulator with various well-defined structures including a buffer management layer, a request allocation layer, an FTL layer, and a hardware module layer [10]. It does not support any RAID organization. To provide a channel-level array architecture, we add an array controller layer on top of the FTL layer. Figure 7 shows a software architecture of the extended simulator, which can be configured to simulate two types of channel-level RAID5 SSDs: a CR5-based SSD and a CR5M-based SSD. The former only needs a RAID5 management module, which includes two parity updating methods RCW and RMW as well as a traditional data reconstruction algorithm DOR [8] (see Figure 7). The latter also needs the RAID5 management module but requires the following changes. First, an MW module is added to work in concert with RCW and RMW (see Algorithm 1). Second, the address mapping function in the FTL layer needs to be redesigned so that data replicas on the mirroring chips can be tracked. Third, it uses MCR rather than DOR for data reconstruction. No matter which type of SSD is simulated (i.e., CR5-based or CR5M-based) the underlying flash management functions like garbage collection and wear-leveling are carried out by a simple page-mapping FTL provided by SSDsim [10]. Note that CR5M can be readily extended to work with any existing complex page-mapping FTL scheme like DFTL [7] or HAT [11]. In what follows, we will use DFTL [7] as an example to show how CR5M can be combined with an existing complex FTL.

To reduce cache overhead, DFTL only caches the most popular mappings in the on-flash SRAM while storing the entire mapping table on flash device itself [7]. It segregates all flash pages into two types: *data pages* and *translation pages*. Data pages contain the real data that is accessed or updated during read/write operations, whereas pages that only store information about logical-to-physical address mappings are called translation pages [7]. To work with DFTL, the following

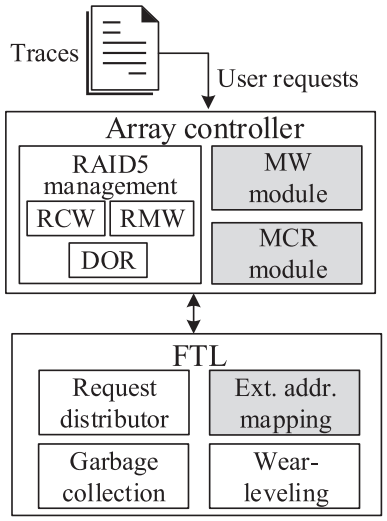


Fig. 7. The simulator software architecture.

changes need to be made in our CR5M design: (1) storing the entire extended mapping table and mirroring table (see Figure 4) in the translation pages on flash; (2) caching some popular mapping entries of the two tables in the on-flash SRAM; (3) adding a Global Translation Directory (GTD) proposed by the authors of DFTL [7] to keep track of all translation pages on flash. Note that GTD is always maintained in the on-flash SRAM. If the mapping information of an incoming request is not presented in the SRAM, then GTD is consulted to find out its corresponding translation page so that the mapping information can be fetched from flash. Detailed information about a DFTL address translation process can be found in Reference [7]. We only consider page-mapping FTLs, because page-level mapping can achieve the best performance among the three mapping schemes (i.e., page-mapping, block-mapping, hybrid) [7].

Although we implement a complete array controller into the SSDsim, only the three modules in grey (see Figure 7) are newly designed. Around 1,000 lines of C code are written to extend SSDsim [10]. The default internal structure configurations and timing parameters of the simulator are summarized in Table 1. GC threshold is set to 30%, which means that GC can be triggered only after the free capacity of an SSD is less than 30 percent of its total capacity. When a data reconstruction algorithm (i.e., DOR or MCR) is evaluated, we assume a data chip failure happens at time instance *failure_time*, which is a tunable parameter of the simulator. During a playback of a trace, the simulator initially runs in the normal mode to process user requests. At a certain time (i.e., *failure_time*), we assume a data chip fails, and thus, the simulator immediately enters into the degrade mode. In this mode, the SSD array controller generates internal read/write requests and uses XOR operations to reconstruct data of the failed data chip onto a replacement chip. Meanwhile, the simulated SSD still serves external user requests. In our experiments of DOR and MCR, the value of *failure_time* is always set to the arrival time of the middle request in a trace.

4.2 Experimental Setup

Five real-world traces are selected to evaluate the performance of CR5M and MCR. Financial2 (hereafter, Fin2) is an I/O trace collected from an OLTP application running at a large financial institution [4]. It is read-dominant. The Build trace is generated from the activities on a Microsoft Build server [36]. In this trace, the number of writes is close to the number of reads and most

Table 1. Simulation Configurations

Parameter	Value
Page size (KB)	4
Number of pages per block	64
Number of blocks per plane	4,096
Number of planes per die	2
Number of dies per chip	2
Number of data chips per channel	4
Number of channels	4
GC threshold	30%
Over-provisioning capacity	10%
Block erase time (ns)	1,500,000
Page write time (ns)	200,000
Page read time (ns)	20,000
Command write time (ns)	25
Register write time (ns)	25

Table 2. The Characteristics of Traces

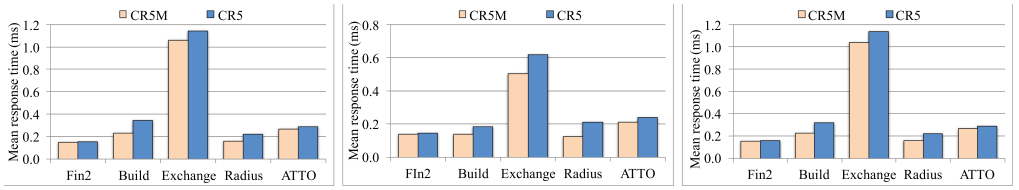
Trace	% of write	Avg. size (KB)	Req/s	Duration (min)	Data traffic (GB)
Fin2	17.7	2	90	683	7.0
Build	45.7	6.5	372	15	2.1
Exchange	46.4	12.5	166	15	1.8
Radius	88.5	6.8	57	35.2	0.8
ATTO	47.5	23.1	792.4	2.5	2.6

Table 3. Configurations of the Three SSDs

Configuration	CR5M	CR5	Capacity
SSD1	$4\ channel * (3 + m + s)\ chip$	$4\ channel * (4 + s)\ chip$	64GB
SSD2	$8\ channel * (3 + m + s)\ chip$	$8\ channel * (4 + s)\ chip$	128GB
SSD3	$4\ channel * (7 + m + s)\ chip$	$4\ channel * (8 + s)\ chip$	128GB

writes are new writes. The Exchange trace is collected from a Microsoft Exchange server and is write-intensive [36]. Both Build and Exchange are collected from a server where a disk array is employed. Radius is collected for the RADIUS authentication server [36]. ATTO is a benchmark gathered from a PC with an NTFS file system by DiskMon [2]. The ATTO trace generates the same amount of I/O requests irrespective of storage capacity and most of the random requests are the accesses to small number of sectors. The characteristics of the traces are summarized in Table 2. The "data traffic" of a trace is the amount of data that have been read/written during the replay of the trace. The purpose of selecting these traces is that CR5M and MCR can be evaluated under various workloads from read-dominant to read-write-even to write-dominant.

To fully evaluate the performance of CR5M on SSDs with various internal architectures (e.g., SSDs with different number of channels), we configure the simulator to simulate three types of SSDs (i.e., SSD1, SSD2, and SSD3) as shown in Table 3. While m stands for a mirroring chip, s represents a spare chip. For example, a CR5M-based SSD1 drive has four channels with each channel having five chips including three data chips, one mirroring chip, and one spare chip.



(a) Experimental results on SSD1 *(b)* Experimental results on SSD2 *(c)* Experimental results on SSD3

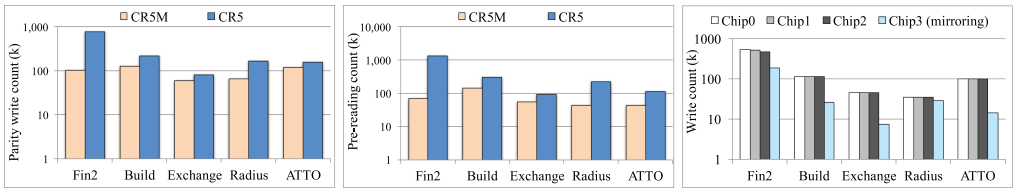
Fig. 8. Performance evaluation of CR5M under real-world traces.

However, a CR5-based SSD1 drive also has four channels and each channel has four data chips and one spare chip (see Table 3). In fact, the difference between the two SSD1 drives is that the former has one less data chip as it has been turned into a mirroring chip. Thus, the user-visible capacity of a CR5M-based SSD1 drive is 25% less than that of a CR5-based SSD1 drive. To make comparisons between CR5M and CR5 fair, the total number of chips of a CR5M-based SSD is always equal to that of a CR5-based SSD. SSD1 is the default SSD configuration for both CR5 and CR5M. In all experiments, 10% of flash memory capacity is preserved as over-provisioning capacity (see Table 1) for bad block management and garbage collection.

4.3 Evaluation of CR5M

Since mean response time is a commonly used metric for evaluating the performance of an SSD, we measure it throughout all experiments in this research. To evaluate the effectiveness of CR5M, we compare it with CR5, which is a standard RAID5 organization applied on the channel level of an SSD. Similar to CR5M, CR5 treats each channel as an independent disk. We also measure number of pre-readings (i.e., pre-reading count) and number of parity writes (i.e., parity write count) for both CR5M and CR5 to find out how many such operations can be diminished by CR5M compared with CR5. A fewer number of such operations is the source of strength of CR5M. In addition, a write distribution among the four chips (i.e., three data chips and one mirroring chip) on channel 0 of a CR5M-SSD1 drive is provided to demonstrate that the mirroring chip receives a fewer number of writes as a data chip. The implication is that it does not become a bottleneck of performance and endurance. To understand the impact of some key features of a workload such as write request percentage and average request size, we also test CR5M under a synthetic benchmark.

Performance comparisons between CR5M and CR5 on the three types of SSDs are shown in Figure 8. Note that on each channel a CR5M-based SSD always has one less data chip than a CR5-based SSD. Therefore, its user-visible SSD capacity and chip-level parallelism are both lower than its CR5 counterpart. However, experimental results shown in Figure 8 show that CR5M outperforms CR5 in all cases, which demonstrates that turning a data chip on each channel into a mirroring chip is worthwhile. This is because the mirroring chips can absorb some small writes so that their associated parity updates can be delayed. Ultimately, the benefits of employing a mirroring chip per channel outweigh the disadvantages caused by a reduced user-visible SSD capacity and chip-level parallelism. On average, CR5M reduces mean response time by 15.8%, 19.4%, and 15.4% under the five real-world traces on SSD1, SSD2, and SSD3, respectively. In particular, CR5M achieves the largest performance improvement over CR5 under Radius (i.e., 40.2% on SSD2 in Figure 8(b)) and Build (i.e., 33% on SSD1 in Figure 8(a)). In the worst case, CR5M can only decrease mean response time by 3.2% (i.e., Fin2 on SSD1 in Figure 8(a)). The reason is that Fin2 is a read-dominant workload, which has few chances for CR5M to execute an MW operation. When the number of channels increases from four to eight (i.e., from SSD1 to SSD2), the performance of both CR5M and CR5 have



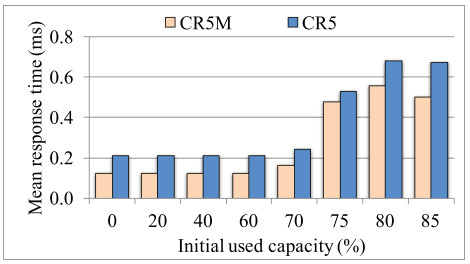
(a) Parity write count *(b)* Pre-reading count *(c)* A write distribution of CR5M

Fig. 9. Comparisons in number of internal operations on SSD1.

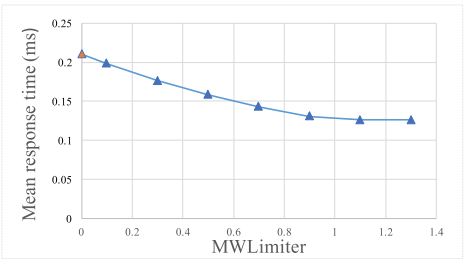
been improved noticeably across all five traces (compare Figures 8(a) and 8(b)). This observation is expected, because the effect of adding more channels in an SSD is equivalent to inserting more disks in a disk array. More importantly, the performance improvement of CR5M over CR5 rises under most traces when the number of channels is increased from four to eight. For instance, the performance improvement of CR5M over CR5 increases from 7% on SSD1 to 18.6% on SSD2 for Exchange. However, comparing results shown in SSD1 (Figure 8(a)) and SSD3 (Figure 8(c)), we observe that the performance of both CR5M and CR5 keeps almost unchanged. The insight is that chip-level parallelism has a very limited impact on SSD performance, which also motivates us to turn a data chip into a mirroring chip in a CR5M architecture.

Compared with CR5, CR5M serves user requests quicker, because it can reduce the number of parity writes and number of pre-reading operations caused by parity updates. In addition, a fewer number of parity writes implies less wear out, which is beneficial to the endurance and reliability of an SSD. To understand how many such operations can be saved by CR5M compared with CR5, we measure the numbers of these two operations for both CR5 and CR5M on SSD1 in Figure 9. Note that all three vertical axises in Figure 9 are in a log scale, because values vary greatly. Figure 9(a) shows that on average CR5M reduces the number of parity writes by 47.9%. In the best case, it diminishes the number of parity writes by 86.8% for Fin2. In the worst case, it can still reduce the number of parity writes by 23.8% for ATTO. Since each MW operation can save one parity write plus one or more pre-reading operations, CR5M can save an even larger number of pre-reading operations (see Figure 9(b)). It at least reduces the number of pre-readings by 42.1% for Exchange. On average, CR5M reduces the number of pre-readings by 66.5% (see Figure 9(b)). To discover whether a mirroring chip receives much more writes than a data chip on the same channel, we also measure the number of writes on each chip of a CR5M-SSD1 drive. Since all four channels of the SSD1 drive exhibit a similar write distribution, Figure 9(c) only shows the write distribution on all working chips (i.e., three data chips and one mirroring chip) of channel 0. From Figure 9(c), one can see that in call cases the number of writes of the mirroring chip is fewer than that of a data chip. For example, the number of writes on the mirroring chip for Exchange is only 16.4% of the average number of writes on a data chip. In the worst case, the number of writes on the mirroring chip is 81.9% of the average number of writes on a data chip for Radius, which is the most write-intensive trace among the five (see Table 2). Figure 9(c) manifests that the mirroring chip does not become a performance/endurance bottleneck for an SSD. Another implication of Figure 9(c) is that recovering a failed mirroring chip takes less time than rebuilding a failed data chip as it contains less amount of data.

To understand the impact of free capacity of an SSD on the performance of CR5M, we conduct a group of experiments to measure the performance of CR5 and CR5M in different initial used capacities (i.e., from 0% to 85%) across the three SSDs shown in Table 3 under the Radius trace. When the initial used capacity of an SSD is set to 0%, Radius is run on an empty SSD. Similarly,



(a) Impact of initial used capacity on CR5M.



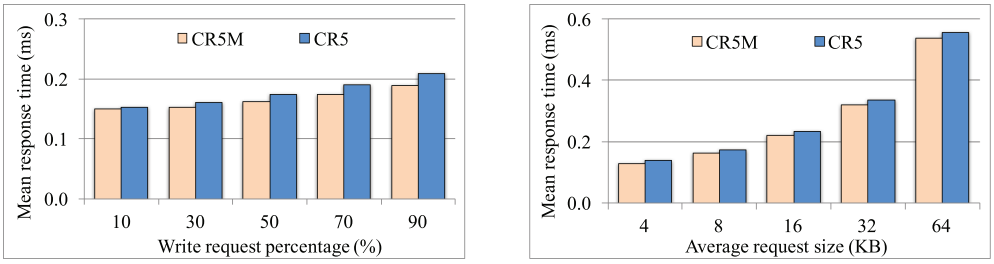
(b) Impact of MWLimiter on CR5M.

Fig. 10. Impact of initial used capacity and MWLimiter on CR5M.

when the initial used capacity is configured to 80%, it starts to run on an SSD whose 80% capacity has been occupied by other user data. We choose Radius, because it is the most write-intensive workload among the five traces. The more write intensive, the more GC operations occur, which makes the impact of initial used capacity on performance more obvious. In addition, the GC threshold is set to 30%, which means that GC can be triggered only after the free capacity of an SSD is less than 30 percent of its total capacity. We only show the results from SSD2 in Figure 10(a), because results from the other two SSDs are very similar to that of SSD2. When the initial used capacity is less than or equal to 60%, we can see that for both CR5 and CR5M their performance remains unchanged. This is because there are still plenty of free pages in the SSD, and thus, GC does not occur. Therefore, shrinking the free capacity of the SSD has little influence on performance. However, after the initial used capacity is larger than 70%, the performance of CR5 and CR5M drops dramatically, because GC operations have been triggered after the GC threshold is reached. We find that CR5M still outperforms CR5 after the initial used capacity is larger than 70%. In general, CR5M gains more performance improvements over CR5 when the SSD has enough free space (i.e., the initial used capacity is no larger than 60%). After GC operations are triggered (i.e., the initial used capacity is between 70% and 85%), on average CR5M outperforms CR5 by 21.4%. We do not test extreme cases where the initial used capacity is equal to 90% or even larger. This is because in real-world scenarios no user would run a workload on an SSD that can only provide 10% or less of its capacity.

To understand the impact of *MWLimiter* on the performance of CR5M, we run Radius on SSD2 under various *MWLimiter* values (i.e., from 0 to 1.3). Again, we choose Radius as it is the most write-intensive workload among the five traces. Figure 10(b) shows the experimental results. When *MWLimiter* is set to 0, CR5M degrades to CR5 as an MW operation can never happen. As *MWLimiter* is enlarged from 0 to 1.1, we notice that the performance of CR5M increases. This is because more MW operations occur, which reduces the parity updating cost. After *MWLimiter* reaches 1.1, further increasing its value is not helpful for performance, because the number of MW operations is also limited by the status of each chip (i.e., "busy/idle"). The experimental results from Figure 10(b) guided us to set 1.1 as the default value of *MWLimiter* in all our experiments.

The impact of two key features of a workload (i.e., write request percentage and average request size) on CR5M is evaluated in Figure 11. Their default values are set to 50% and 8KB, respectively. On average 100 requests arrived per second and there are totally 100,000 requests in the synthetic benchmark. The request arrival time and request size all follow a uniform distribution. All experiments in this group are conducted on an SSD1 (see Table 3). Figure 11(a) shows that the performance improvement of CR5M over CR5 increases from 1.5% to 9.8% when the write request



(a) Write request percentage. *(b)* Average request size.

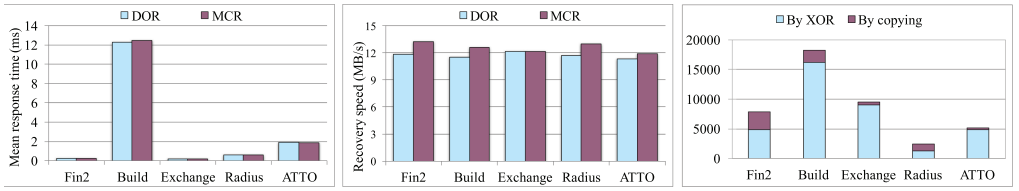
Fig. 11. Impact of two key workload characteristics.

percentage enlarges from 10% to 90%. The reason is that CR5M can save more internal operations (i.e., pre-readings and parity writes) caused by parity updates when a workload becomes more write-intensive. When the average request size increases from 4KB to 64KB, the performance of both CR5M and CR5 decreases as shown in Figure 11(b). This result is understandable as a larger request requires more time to serve. However, the performance improvement of CR5M over CR5 decreases from 7.4% to 3.1% when the average request size enlarges from 4 to 64KB. The reason is that when the sizes of most requests are larger than the stripe width there are many full-stripe sub-requests, which cannot utilize MW operations (see Algorithm 1).

4.4 Evaluation of MCR

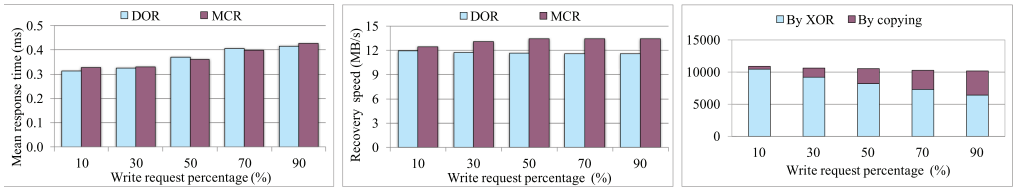
To evaluate MCR, we compare it with DOR, a widely used traditional data reconstruction algorithm. Most existing data reconstruction algorithms are either not appropriate to a channel-level RAID-structured single SSD (e.g., MICRO [44], PRO [39], and s^2-RAID [40]) or not designed for RAID5 (e.g., PDB [21]). For example, MICRO [44] relies on a multi-level cache architecture, which does not exist in an SSD. s^2-RAID [40] requires multiple spare disks in an HDD array. For a channel-level RAID5 SSD, paying multiple spare channels for a fast data recovery might be too expensive to be acceptable in industry. However, the basic idea of PRO is to rebuild popular data first, which is contradictory to the fact that in a CR5M-based SSD a popular page of data on a failed data chip most likely has a replica on the mirroring chip, and thus, do not need to be recovered first as we explained in the last paragraph of Section 3.2.2. Since recovering a failed mirroring chip is much faster than rebuilding a failed data chip, in this section, we only compare MCR with DOR when a data chip fails.

We measure data recovery speed (i.e., the amount of lost data that can be recovered per second) and mean response time (MRT) during reconstruction throughout the experiments in this section. In addition, we also examine the number of strips that are reconstructed through an XOR operation and the number of data strips that are recovered by copying a replica from the mirroring chip to a replacement chip in CR5M. All experiments in this section have been conducted on an SSD1 configuration (see Table 3). Figure 12(a) shows mean response times during reconstruction for both DOR and MCR under the five real-world traces. We find that MCR delivers a mean response time during reconstruction very similar to DOR. The reason is that similar to DOR, MCR insert a pre-read request to a channel once the channel is idle. Thus, a channel keeps busy all the time in both DOR and MCR. Therefore, user requests will be treated similarly in two data reconstruction schemes, which results in a similar mean response time. In the worst case, MCR only increases mean response time during reconstruction by 0.6% under Build compared with DOR. As for the best case, MCR reduces mean response time during reconstruction by 3.6% under Exchange. On



(a) MRT during reconstruction *(b)* Data recovery speed *(c)* Reconstructed strip count

Fig. 12. Performance evaluation of MCR under real-world traces.

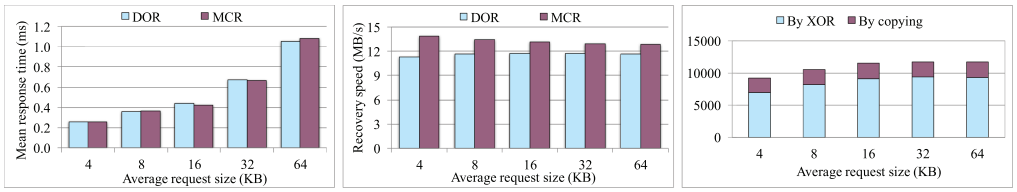


(a) MRT during reconstruction *(b)* Data recovery speed *(c)* Reconstructed strip count

Fig. 13. Impact of write percentage on data reconstruction.

average, MCR improves performance in terms of mean response time during reconstruction by 0.9%, which can be explained by random variation. Figure 12(b) shows that MCR always achieves a higher data recovery speed among all five traces compared with DOR. On average, MCR increase data recovery speed by 7.4%. In the best scenario, MCR improves data recovery speed by 11.5% under Fin2. Results from Figure 12(b) demonstrate that MCR can further shrink the window of vulnerability. Figure 12(c) illustrates that among all reconstructed strips (either a data strip or a parity strip) during a data reconstruction process how many of them are recovered by copying a replica from the mirroring chip to a replacement chip. We find that at least 4.6% (Exchange) of reconstructed data can be recovered by directly copying from the mirroring chip. In the case of Radius, 45.9% of reconstructed data can be recovered without any XOR operations. Note that copying one strip of data from the mirroring chip to a replacement chip on the same channel demands one internal read (from the mirroring chip to the RAID controller) and one internal write (from the RAID controller to a replacement chip). However, recovering a strip in DOR needs multiple parallel internal reads, multiple XOR operations, and one internal write (see phase one in Figure 6). Considering that the CPU time spent on the multiple XOR operations is trivial compared with the read/write time of one strip of data, the saving of a directly copying is not substantial. That is why on average MCR only improve data recovery speed by 7.5%.

Similar to CR5M evaluation, we also test the impact of two major workload characteristics on the performance of MCR. We use the same synthetic benchmark described in Section 4.3. Again, the value of *failure_time* is always set to the arrival time of the middle request in the benchmark. Figure 13 shows the impact of write request percentage on the performance of MCR and DOR. The general trend shown in Figure 13(a) is that mean response time during reconstruction increases for both MCR and DOR as the write request percentage escalates. Another observation is that the mean response time of MCR is very close to that of DOR. On average, the mean response time during reconstruction of MCR is only 0.8% longer than that of DOR (see Figure 13(a)). However, Figure 13(b) demonstrates that MCR on average can improve data recovery speed by 12.5%. In particular, when 90% requests are writes MCR improves data recovery speed by 15.9%. The



(a) MRT during reconstruction *(b)* Data recovery speed *(c)* Reconstructed strip count

Fig. 14. Impact of average request size on data reconstruction.

implication is that MCR exhibits its full strength under a write-intensive workload. Figure 13(c) shows that the percentage of data strips recovered by directly copying a replica from the mirroring chip to a replacement chip increases when the percentage of write requests enlarges. When the percentage of write requests is 10%, only 4.7% reconstructed strips are recovered through directly copying. However, this number increases to 36.5% when the percentage of write requests becomes 90%. Obviously, with more write requests, more MW operations can be performed when an SSD is the normal mode. Thus, more data can be recovered by directly copying when the SSD is in the degraded mode, which accelerates data recovery speed. Figure 14 shows the impact of average request size on the performance of MCR and DOR. When average request size enlarges from 4KB to 64KB, Figure 14(a) shows that the mean response times during reconstruction of both MCR and DOR substantially increase. In the best case, when the average request size is 16KB MCR delivers a mean response time shorter than that of DOR by 3.3%. In the worst case, the mean response time of MCR is longer than that of DOR by 2.5% when average request size is 64KB. Still, in terms of data recovery speed, MCR always outperforms DOR as shown in Figure 14(b). On average, MCR improves data recovery speed by 10.1% compared with DOR%. MCR achieves its largest data recovery speed improvement over DOR by 22.6% when average request size is 4KB (see Figure 14(b)). The results shown in Figure 14(c) demonstrate that the percentage of number of reconstructed strips by directly copying decreases from 24.6% to 20.8% as average request size increases from 4 to 64KB. The reason is that when average request size increases there are more full-stripe sub-requests, which cannot use an MW operation (see Algorithm 1).

5 CONCLUSIONS

An off-the-shelf SSD may not be able to satisfy the data reliability requirements demanded by a safety-critical mobile application. Thus, applying a data redundancy mechanism like RAID at a higher level within an SSD becomes necessary. In this article, we first propose a new RAID5 architecture called CR5M at the channel-level of a single SSD. Next, an associated data reconstruction strategy called MCR is developed to further shrink the window of vulnerability. Experimental results demonstrate their effectiveness. A CR5M-based SSD requires the same amount of flash chips as a CR5-based SSD does (see Table 3). However, it can deliver a better performance/endurance and data reliability for a safety-critical mobile application.

REFERENCES

[1] Guillermo A. Alvarez, Walter A. Burkhard, Larry J. Stockmeyer, and Flaviu Cristian. 1998. Declustered disk array architectures with optimal and near-optimal parallelism. In *ACM SIGARCH Computer Architecture News*, Vol. 26, 3. IEEE Computer Society, 109–120.

[2] ATTO. 2017. Disk benchmarks. Retrieved from https://www.attotech.com/disk-benchmark/.

[3] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi. 2010. Differential RAID: Rethinking RAID for SSD reliability. *ACM Trans. Stor.* 6, 2 (2010), 4.

[4] K. Bates and B. McNutt. 2002. OLTP Application I/O. Retrieved from http://traces.cs.umass.edu/index.php/Storage/Storage.
[5] Kevin M. Greenan, Darrell D. E. Long, Ethan L. Miller, Thomas Schwarz, and Avani Wildani. 2009. Building flexible, fault-tolerant flash-based storage systems. In *Proceedings of the Workshop on Hot Topics in Dependable Systems (HotDep'09)*, vol. 9.
[6] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The bleak future of NAND flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. USENIX Association, 2.
[7] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. 2009. *DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings*. Vol. 44. ACM.
[8] Mark Holland. 1994. *On-line Data Reconstruction in Redundant Disk Arrays*. Ph.D. Dissertation. Citeseer.
[9] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. 1993. Fast, on-line failure recovery in redundant disk arrays. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS'93)*. IEEE, 422–431.
[10] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Hao Luo, and Shuping Zhang. 2011. Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity. In *Proceedings of the International Conference on Supercomputing*. ACM, 96–107.
[11] Yang Hu, Hong Jiang, Dan Feng, Lei Tian, Shuping Zhang, Jingning Liu, Wei Tong, Yi Qin, and Liuzheng Wang. 2010. Achieving page-mapping FTL performance at block-mapping FTL cost by hiding address translation. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–12.
[12] Soojun Im and Dongkun Shin. 2010. Delayed partial parity scheme for reliable and high-performance flash memory SSD. In *Proceedings of the IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. IEEE, 1–6.
[13] 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.
[14] Jung-Ho Kim, Edward Lee, Jang-Young Choi, Daewoo Lee, and Samyeul Noh. 2016. Chip-Level RAID with flexible stripe size and parity placementfor enhanced SSD reliability. *IEEE Trans. Comput.* 65, 4 (2016), 1116–1130.
[15] 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 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST'11)*. IEEE, 1–12.
[16] John C. Knight. 2002. Safety critical systems: Challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*. IEEE, 547–550.
[17] Steve Larrivee. 2015. Solid state drive primer # 8—Controller architecture—Channels and banks. Retrieved from https://www.cactus-tech.com/resources/blog/details/solid-state-drive-primer-8-controller-architecture-channels-and-banks.
[18] Sehwan Lee, Bitna Lee, Kern Koh, and Hyokyung Bahn. 2011. A lifespan-aware reliability scheme for RAID-based flash storage. In *Proceedings of the 2011 ACM Symposium on Applied Computing*. ACM, 374–379.
[19] Yangsup Lee, Sanghyuk Jung, and Yong Ho Song. 2009. FRA: A flash-aware redundancy array of flash storage devices. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. ACM, 163–172.
[20] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, and Tong Zhang. 2015. How much can data compressibility help to improve NAND flash memory lifetime? In *Proceedings of the Conference on File and Storage Technologies (FAST'15)*. 227–240.
[21] Feng Liu, Wen Pan, Tao Xie, Yanyan Gao, and Yiming Ouyang. 2013. PDB: A reliability-driven data reconstruction strategy based on popular data backup for RAID4 SSD arrays. In *Algorithms and Architectures for Parallel Processing*. Springer, 87–100.
[22] Michael S. MacFadden, Richard Shelby, and Tao Xie. 2015. SIRF-1: Enhancing reliability of single flash SSD through internal mirroring for mission-critical mobile applications. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'15)*. IEEE, 343–351.
[23] Lucas Mearian. 2016. SSD prices plummet again, close in on HDDs. Retrieved from http://www.pcworld.com/article/3040591/storage/ssd-prices-plummet-again-close-in-on-hdds.html.
[24] Jai Menon. 1995. A performance comparison of RAID-5 and log-structured arrays. In *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*. IEEE, 167–178.
[25] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM, 177–190.
[26] Micron. 2015. M500IT mSATA NAND flash SSD data sheet. Retrieved from https://www.micron.com/resource-details/1414bd8a-ca68-4909-9f9f-9beae7cc59b0.
[27] Richard R. Muntz and John C. S. Lui. 1990. *Performance Analysis of Disk Arrays Under Failure*. Computer Science Department, University of California.

[28] Samsung on Amazon. 2018. Samsung 840 EVO 500GB 2.5-inch SATA III internal SSD (MZ-7TE500BW). Retrieved from https://www.amazon.com/DISCONTINUED-Samsung-2-5-Inch-Internal-MZ-7TE500BW/dp/B00E3W19MO.

[29] Wen Pan, Feng Liu, Tao Xie, Yanyan Gao, Yiming Ouyang, and Tian Chen. 2013. Spd-raid4: Splitting parity disk for raid4 structured parallel ssd arrays. In *Proceedings of the 10th International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC-EUC'13)*. IEEE, 9–16.

[30] Kwanghee Park, Dong-Hwan Lee, Youngjoo Woo, Geunhyung Lee, Ju-Hong Lee, and Deok-Hwan Kim. 2009. Reliability and performance enhancement technique for SSD array storage system using RAID mechanism. In *Proceedings of the 9th International Symposium on Communications and Information Technology (ISCIT'09)*. IEEE, 140–145.

[31] Chris Ramseyer. 2016. Intel 600p series SSD review. Retrieved from http://www.tomshardware.com/reviews/intel-600p-series-ssd-review,4738.html.

[32] Jacob Rosen and Blake Hannaford. 2006. Doc at a distance. *IEEE Spectrum* 43, 10 (2006), 34–39.

[33] Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? In *Proceedings of the Conference on File and Storage Technologies (FAST'07)*, Vol. 7. 1–16.

[34] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*. 67–80.

[35] Joseph M. Smith. 2011. The doctor will see you ALWAYS. *IEEE Spectrum* 48 (2011), 56–62. Issue 10.

[36] SNIA. 2011. Microsoft Production Server Traces. Retrieved from http://iotta.snia.org/traces/158.

[37] Trefis Team. 2014. Here's why DRAM prices will decline in the future. Retrieved from https://www.forbes.com/sites/greatspeculations/2014/05/05/heres-why-dram-prices-will-decline-in-the-future/#34164d222694.

[38] Techopedia. 2003. Mobile data center. Retrieved from http://www.naat.com/Disaster%20Recovery/mobile_datacenter.htm.

[39] Lei Tian, Dan Feng, Hong Jiang, Ke Zhou, Lingfang Zeng, Jianxi Chen, Zhikun Wang, and Zhenlei Song. 2007. PRO: A popularity-based multi-threaded reconstruction optimization for RAID-structured storage systems. In *Proceedings of the Conference on File and Storage Technologies (FAST'07)*, Vol. 7. 277–290.

[40] Jiguang Wan, Jibin Wang, Changsheng Xie, and Qing Yang. 2014. s2-RAID: Parallel RAID architecture for fast data recovery. *IEEE Trans. Parallel Distrib. Syst.* 25, 6 (2014), 1638–1647.

[41] Wei Wang, Tao Xie, and Abhinav Sharma. 2016. SWANS: An interdisk wear-leveling strategy for RAID-0 structured SSD arrays. *ACM Trans. Stor.* 12, 3 (2016), 10.

[42] Yu Wang, Wei Wang, Tao Xie, Wen Pan, Yanyan Gao, and Yiming Ouyang. 2014. CR5M: A mirroring-powered channel-RAID5 architecture for an SSD. In *Proceedings of the 30th Symposium on Mass Storage Systems and Technologies (MSST'14)*. IEEE, 1–10.

[43] Suzhen Wu, Bo Mao, Xiaolan Chen, and Hong Jiang. 2016. LDM: Log disk mirroring with improved performance and reliability for SSD-Based disk arrays. *ACM Trans. Stor.* 12, 4 (2016), 22.

[44] Tao Xie and Hui Wang. 2008. Micro: A multilevel caching-based reconstruction optimization for mobile storage systems. *IEEE Trans. Comput.* 57, 10 (2008), 1386–1398.

Received September 2017; revised April 2018; accepted April 2018