Enabling TRIM Support in SSD RAIDs

Informatik Preprint CS-05-11, ISSN 0944-5900
Department of Computer Science, University of Rostock
September 2011

Nikolaus Jeremic[1], Gero Mühl[1], Anselm Busse[2], and Jan Richling[2]

[1] Architecture of Application Systems Group,
University of Rostock, Germany
{nikolaus.jeremic,gero.muehl}@uni-rostock.de
[2] Communication and Operating Systems Group,
Berlin University of Technology, Germany
{abusse,richling}@cs.tu-berlin.de

Abstract. Deploying solid state drives (SSDs) in RAID configurations can leverage the performance of persistent storage systems into a new dimension. However, the Achilles' heel of SSDs and, therefore, also SSD RAIDs is their write performance, especially for small random requests. To prevent a degradation of the write throughput, it is important to maintain an adequate amount of free flash capacity. This can be ensured by over-provisioning and additionally encouraged by notifying the utilized SSDs of storage space no longer required (e.g., because the occupying file was deleted) using the `TRIM` command. However, many current hardware and software RAID implementations do not propagate `TRIM` commands to their member drives at all and especially not in RAID setups using parity information to increase the reliability. This leads to a severe limitation of the intention behind the `TRIM` command because parity-based RAIDs are part of many persistent storage systems.
In this report, we discuss the issues of enabling `TRIM` support in SSD RAIDs and explain how `TRIM` can be incorporated into RAID implementations for commonly used RAID levels.

1 Introduction

A single *solid state drive (SSD)* can today exhibit a performance that was previously only possible with a *Redundant Array of Independent Disks (RAID)* setup consisting of a bunch of enterprise-class SAS hard disk drives. By deploying SSDs in RAID configurations, the performance can be elevated to even much higher levels. SSDs employ *NAND-flash memory*, whose technological properties require to perform updates of the stored data out-of-place, meaning that the new content is stored in another physical part of the memory. This way of processing data updates necessitates to reclaim the storage capacity occupied by the outdated data when free storage space is becoming scarce. However, most filesystems only update their metadata when a file is deleted and do not perform further modifications of the allocated storage space until it is reused in

order to store new data. Due to this fact, an SSD will normally have no information about which part of its storage space contains obsolete data and could, therefore, be reclaimed. Consequently, the storage capacity considered to be free will be smaller than it is, which is unfavorable because the amount of free flash capacity is a major factor determining write performance, in particular for small random writes. In order to tackle this issue, the `TRIM` command was introduced for SSDs that are equipped with the *Serial Advanced Technology Attachment (SATA)* interface. In systems using the *Serial Attached SCSI (SAS)* interface a similar functionality is offered through the commands `UNMAP` and `WRITE SAME`.

The `TRIM` command allows to notify an SSD about storage space whose content is obsolete. This works without stint provided that no further intermediate software layers, like software RAID drivers or logical volume managers, are interposed. To the best of our knowledge the *Device Mapper RAID* in *Linux* is the only RAID implementation, which currently supports `TRIM` commands, but only for parity-less RAID configurations (RAID 0, RAID 1 and RAID 10). The authors of this report are not aware of any RAID implementation that supports `TRIM` in parity-based RAID setups. Currently available hardware RAID controllers also do not propagate `TRIM` commands to the member drives. Due to lacking support, the `TRIM` command provides at present no advantage in many persistent storage systems that are in use.

This report focuses on SATA-based storage systems, where hardware or software RAID implementations are interposed between a filesystem and the SSD devices. The objective is to incorporate the support for `TRIM` commands into such storage systems in order to improve the write throughput by facilitating the garbage collection mechanism of the deployed SSDs. The `TRIM` command is specified by the Serial ATA standard [14] since it was proposed back in 2007 [13]. Each `TRIM` command comprises at least one *Logical Block Addressing (LBA)* range consisting of a starting sector and the number of sectors, which can be considered as unallocated. A `TRIM` command issued to a RAID device must be translated similarly to write requests to be properly processed by all affected member drives. Additionally, the parity information has to be maintained in RAID configurations like RAID 5 or RAID 6, which is potentially more complicated than for write requests due to the implementation of the `TRIM` command inside an SSD.

The remainder of this report is structured as follows. We provide the technical background on SSDs, the `TRIM` command and RAIDs in Sect. 2. In Sect. 3, we explain how `TRIM` can be incorporated into common RAID organizations. We consider the integration of the `TRIM` command into operating systems as well as the current technical limitations of the `TRIM` command. Finally, we discuss the introduced approach for `TRIM` support in SSD RAIDs in Sect. 4 and give an overview of our further work.

2 Technical Background

This section provides the required technical foundations of *Solid State Drives (SSDs)*, the TRIM command as well as of the *Redundant Array Of Independent Disks (RAID)* technique.

2.1 Solid State Drives

SSDs employ NAND-flash memory for persistent storage of data. NAND-flash memory is internally organized in *blocks* comprising multiple *pages* that represent the smallest writable data portions. The capacity of a single flash page ranges from 512 bytes to several kilobytes and one flash block consists usually of 64 or 128 flash pages. Flash memory exposes two crucial properties: Each flash memory cell has to be erased before programming and the number of erase cycles is limited. The actual limit depends on the particular type of flash memory and can range from several thousands to a million erase cycles. Due to these flash memory characteristics, a NAND-flash page has also to be erased before data can be written to it. Erases are performed on whole NAND-flash block erasing all its pages in parallel. A block erasure takes significantly longer than programming or reading of a page. For brevity, we will use the term "flash memory" instead of "NAND-flash memory" for the remainder of this report.

The limited endurance of flash memory is addressed by *wear leveling*, which ensures that erase operations are balanced over all blocks. When the content of a single flash page would have to be changed in-place, it would be necessary to save the data of all the other pages, erase the flash block, write the saved data back and write the new content to the mentioned page. To avoid this unnecessary burden, the new content of the page is written into an empty page and the old page is marked as invalid. This requires to keep information about the mapping between logical addresses of data and the corresponding physical flash pages. This mapping is managed by the *flash translation layer (FTL)* usually implemented in the firmware of the SSD controller. Consequently, each flash page is always in one of these three states: empty, valid or invalid. An *empty* page has been erased and is, thus, writable. A *valid* page contains valid data that is passed to the host if the page is read. An *invalid* page contains outdated data. To avoid exhausting all empty pages, the invalidated pages have to be reclaimed. This is done by the *garbage collection* mechanism, which is usually also implemented in the firmware of the SSD controller. It first copies the contents of valid pages within a block to another block and subsequently erases the particular block. For this reason, the garbage collection incurs additional page writes within the SSD compared to the write operations issued by the host, leading to *write amplification* [3]. In order to deliver a high write throughput, write amplification must be limited. One effective approach is to guarantee a substantial amount of free flash pages by *over-provisioning*. This approach increases the *spare capacity* that equals to the difference between the physical capacity of the flash memory of an SSD and the capacity advertised to the host's operating system.

2.2 The TRIM Command

In contrast to SSDs, *Hard Disk Drives (HDDs)* can update data in place without a significant reduction of their life time and without performing additional read or write operations decreasing the throughput of the drive. This characteristic of HDDs has led to the situation that filesystems usually only update their metadata when a file is deleted leaving the content of the corresponding storage space untouched until it is allocated to another file. While this remains without consequences for HDDs, it prevents SSDs from considering the corresponding flash pages as invalid (cf. Sect. 2.1) and reclaiming them to increase the amount of empty flash pages. The SSD only becomes aware that the page's data is invalid, when the corresponding logical address is referenced next time by a write request.

For SSDs using the SATA interface as interconnection to host bus adapters, the `TRIM` command was proposed back in 2007 [13] in order to rectify this situation. Since then, the `TRIM` command is incorporated into the SATA standard [12, 14]. This command allows the filesystem to notify the SSD controller of a logical address range whose content can be considered as obsolete. Within the SATA standard, addresses are specified according to the *Logical Block Addressing (LBA)* scheme. The LBA scheme addresses the storage capacity of a device by a linear mapping of sectors, whereby a sector represents the smallest addressable unit. With each `TRIM` command, at least one *LBA range* has to be provided, where each LBA range consists of a starting address (LBA sector number) and the number of sectors that shall be trimmed beginning with the given LBA sector number [14]. The maximum number of LBA ranges allowed per `TRIM` command depends on the used SSD, while the range length is limited to $65,535$ sectors.

The main drawback of the `TRIM` command is its interaction with commands that rely on *Native Command Queuing (NCQ)*, which is an important feature introduced by the SATA standard in order to increase the performance of storage devices. NCQ permits to issue up to 32 read or write commands simultaneously to a SATA device. It was originally introduced to provide SATA HDDs with the ability to reorder read and write requests within the command queue to minimize the read/write head movement and, hence, increase the throughput. This technique is also beneficial for SSDs because it permits to utilize more than one flash memory chip at once by serving multiple read or write requests concurrently. However, the `TRIM` command cannot be queued using NCQ and would cause a cancellation of queued commands when issued while NCQ-enabled commands are processed [12, 14]. This introduces a performance penalty when a series of queued reads or writes is interrupted by `TRIM` commands. Each `TRIM` command forces the device driver to issue the `TRIM` solely, wait for its completion, and then continue to issue further queueable requests to the device. This decreases the degree of parallelism in processing I/O requests.

The integration of the `TRIM` command into current operating systems happens mainly through the filesystem drivers but also through the swap space drivers. In order to benefit from the `TRIM` command, filesystems have to be aware of

the possibility to use TRIM and make use of it. Filesystems usually make use of the TRIM command when a particular part of the storage space does no longer contain needed data, e.g., when a file has been deleted. In this case, the set of LBA ranges that were assigned to the deleted file is trimmed. This can be performed immediately or postponed in order to trim a larger part of storage space at a later time, e.g., when more changes of the filesystem content have been accomplished.

2.3 RAID

RAIDs involve the aggregation of multiple drives in order to increase the performance, reliability or the storage capacity of the resulting logical drive. Patterson et al. [11] introduced RAIDs initially. Since then, further RAID organizations and optimizations of the existing organizations have been introduced [1, 8, 10]. Performance analyzes and surveys can, for example, be found in [2, 6]. The key concepts incorporated into RAIDs are *data striping*, *data mirroring* and the maintenance of *parity information*.

Data striping aggregates the local address ranges of the member drives and decomposes data portions to scatter them across the member drives to increase the throughput. The data is organized in logically continuous data portions named *stripes*, which comprise multiple *chunks* distributed over the member drives. A disadvantage of pure striping is that it suffers more likely from a data loss since for multiple drives, a drive failure becomes more probable. Data mirroring is intended to reduce the chance of a data loss due to a drive failure by keeping multiple copies of the data. This comes with a lower throughput and usable storage capacity. Parity information can be used to improve throughput and reliability. This is achieved by augmenting data striping with parity information in order to be able to recover data when drives fail or when a sector becomes inaccessible. RAID configurations are differentiated according to their level. Commonly used RAID levels are RAID 0 (pure data striping), RAID 1 (pure data mirroring), RAID 5 and RAID 6 (data striping with one or two parities, respectively). Additionally, multiple RAID devices can be combined to a hierarchical RAID (e.g., RAID 10, RAID 50 or RAID 60).

3 Incorporating TRIM support into RAIDs

In this section, we first describe the translation of TRIM commands and the processing of parity updates in hardware and software RAID implementations. Both can be necessary in order to appropriately handle TRIM commands issued to a RAID device. In the second part of this section, we then consider the propagation of TRIM commands in RAID organizations.

3.1 Handling the TRIM command in RAIDs

From the perspective of a RAID device (regardless of whether a hardware or a software RAID implementation is used), a TRIM command has to be handled

similar to a write request because a `TRIM` command issued to a RAID device has to be translated into one or more `TRIM` commands that must be sent to the affected member drives. In contrast to a write request that provides a single LBA range, a `TRIM` command can include multiple LBA ranges, where each of them consists of a starting address (LBA sector number) and a number of sectors to be trimmed beginning with the given LBA sector number. Depending on the used RAID level, an address translation and a parity information update may also be required in order to serve a `TRIM` command. In the following subsections, we explain how `TRIM` commands can be handled in RAIDs considering different RAID levels. We start with the parity-less RAID levels (RAID 0 and RAID 1), continue with the parity-based RAID levels (RAID 5 and RAID 6) and finally discuss hierarchical RAID levels (e.g., RAID 10).

RAID implementations will use a part of the available storage capacity for metadata in certain RAID setups. This leads to an overall storage capacity that is smaller than the total capacity of the underlying devices. Storing the RAID metadata on the member drives necessitates shifting the local address range. Furthermore, some RAID implementations permit to use partitions of the member drives, which causes a further shift of the address range. However, these considerations involve shifting addresses by a fixed offset only and are, therefore, excluded in the following description of `TRIM` translation.

Algorithm 1 Algorithm for translation of LBA range to chunk ranges.

Input:
- LBA range $r = \{start, length\}$
- chunk size c_{size} in sectors

Output: List L_c of chunk ranges.

$L_c := \{\}$
$sectors := r.length$
$chunk_length := c_{\text{size}}$
$position := r.start$
while $sectors > 0$ **do**
  $offset := position \mod c_{\text{size}}$
  if $sectors \geq (c_{\text{size}} - offset)$ **then**
    $chunk_length := c_{\text{size}} - offset$
  else
    $chunk_length := sectors$
  end if
  $c := (position \div c_{\text{size}} : offset : chunk_length)$
  $L_c := L_c \cup \{c\}$
  $position := position + chunk_length$
  $sectors := sectors - chunk_length$
end while

Parity-Less RAID Levels. The parity-less RAID levels RAID 0 and RAID 1 pursue two different goals: while RAID 0 uses data striping and focuses on increasing performance, RAID 1 uses data mirroring targeting mainly at a higher reliability. In RAID 0 configurations, the (global) LBA range of the RAID device is divided in fixed-size chunks (also called *stripe units*) that are distributed over multiple drives in round-robin manner. Hence, a LBA range that is provided with a `TRIM` command can span one or more chunks. In addition to this, one or two of the spanned chunks can be covered only partially, i.e., the LBA range provided with a `TRIM` command may not be aligned to the chunk boundaries. For RAID configurations that employ data striping such as RAID 0, the translation of a LBA range supplied with a `TRIM` command can be carried out in two steps. In the first step, the affected chunks are determined by decomposing the given LBA range. This can be performed according to the algorithm 1 that maps a LBA range to chunk ranges. The second step is the mapping of chunk ranges to local LBA ranges of the member drives. A chunk range $(num : start : len)$ comprises the chunk number given by *num*, a starting sector *start* and the range length *len* in sectors.

After the identification of the affected chunks, the calculated chunk ranges are mapped to the underlying drives in order to accomplish the `TRIM` command on the RAID device. RAID 0 configurations maintain no parity information. For this reason, all chunks contain data. This permits a straight-forward mapping of chunk ranges to LBA ranges on underlying devices: For a RAID 0 configuration with n drives and a chunk range $(num : start : len)$, the corresponding drive number equates to $num \mod n$. The starting sector is computed as $[(num \div n) \cdot c_{\text{size}}] + start$, where c_{size} corresponds to the number of sectors per chunk and $num \div n$ to the stripe number. Please note that $\div$ denotes an integer division. An example of `TRIM` translation is shown in Fig. 1.

To support `TRIM` in RAID 1 configurations, a single `TRIM` command issued to a RAID device has to be duplicated and sent to each member drive. An address translation like for RAID 0 configuration is not necessary as the address ranges of the mirrors are not aggregated. However, as already mentioned, it can be necessary to shift the addresses in real RAID systems, which is neglected in the following example. The example shown in Fig. 2 illustrates the duplication of a `TRIM` command in a RAID 1 organization with 4 drives used for quadruple data mirroring.

Parity-Based RAID Levels. The parity-based RAID levels RAID 5 and RAID 6 protect data with one or two parities, respectively. Parity-based RAID levels with more than two parities are possibly needed in the future for data reliability reasons [8]. For these levels, a similar approach as depicted in the following can be used.

Similar to RAID 0, parity-based RAID configurations rely on data striping and, thus, necessitate an address translation to support `TRIM` commands issued to the RAID device. Furthermore, the parity information has also to be updated after issuing `TRIM` commands to the underlying drives to keep the parity infor-

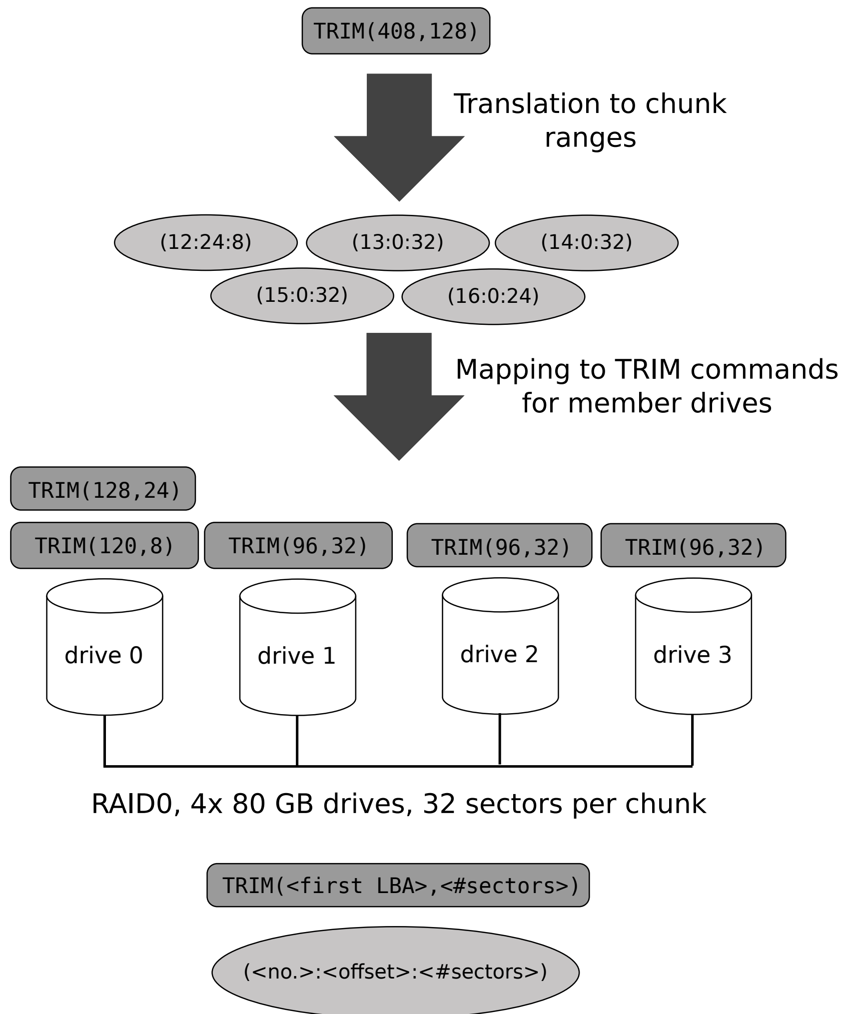


Fig. 1. Exemplary `TRIM` translation in a RAID 0 configuration with 4 drives.

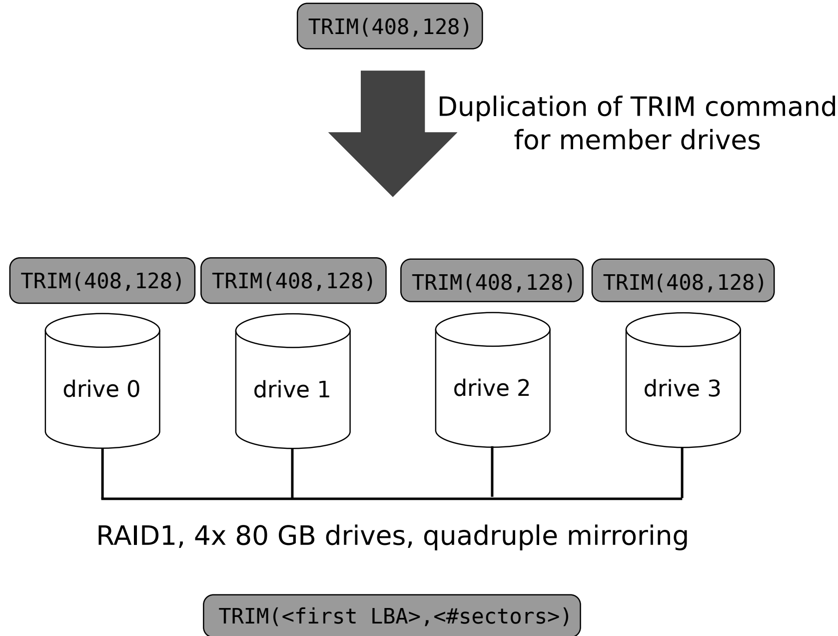


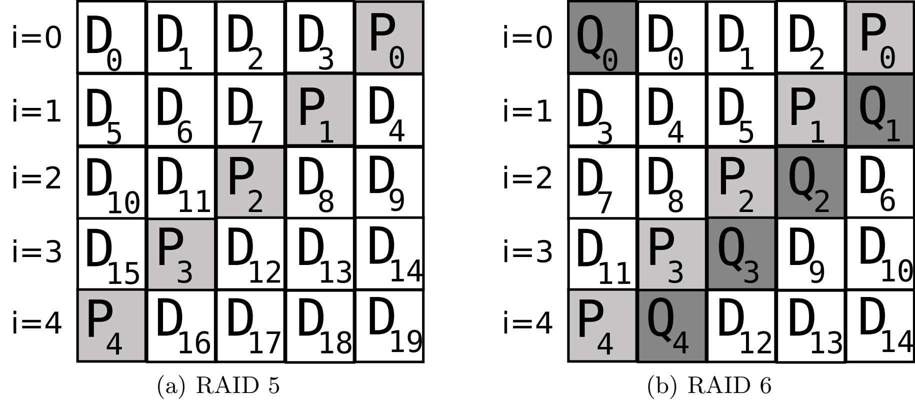
Fig. 2. Exemplary `TRIM` translation in a RAID 1 configuration with quadruple data mirroring.

mation consistent with the data. In the following, we first describe the address translation and deal then with the required updates of the parity information.

The first step of the address translation is identical to RAID 0 configurations explained in the previous paragraph, i.e., for each `TRIM` command all affected chunks can be determined using algorithm 1. For the second step, the parity information has also to be considered. For example, in a RAID 5 setup only one chunk per stripe carries parity information, while RAID 6 configurations possess two parity chunks per stripe. Besides the different number of parity chunks per stripe, several parity placement layouts have been proposed. Each parity placement layout can be characterized by a *parity-placement-function*. The location of the data chunks is then described by an appropriate *data-mapping-function*. Widely used parity-placement-functions in conjunction with the corresponding data-mapping-functions can be found in [7]. According to the authors of [5] and [7], the left-symmetric parity placement and its variants offer the best performance for RAID 5 configurations employing HDDs. As a consequence, this scheme (Fig. 3) is used by default in current RAID implementations, e.g., in Linux Software RAID.

In this report, we examine the translation of a `TRIM` command in parity-based RAID configurations using the left-symmetric parity placement as an example. The following description of mapping the chunk ranges to LBA ranges on un-

derlying drives can be easily transferred to other parity placement algorithms by using the appropriate *data-mapping-function* and *parity-placement-function*.



(a) RAID 5 (b) RAID 6

Fig. 3. Left-symmetric parity layout in a RAID 5 (a) and in a RAID 6 (b), both comprising 5 drives. Each column corresponds to a drive, while data chunks are denoted by D_k. Parity chunks for the stripe i (row) are denoted by P_i in a RAID 5 (a) as well as by P_i and Q_i in a RAID 6 (b).

Consider a RAID 5 configuration with n drives and $p = 1$ parity chunks per stripe that uses left-symmetric parity placement. To determine the corresponding drive for a chunk range $(num : start : len)$, we first compute the stripe number $m = num \div (n - p)$. The next step is to calculate the index *p_drive* of the drive that holds the parity chunk for stripe m. This is equal to $p_drive = (n - p) - (m \mod n)$. Then, the index of the drive containing the particular data chunk corresponds to $d_drive = (p_drive + (num \mod (n - p)) + 1) \mod n$. The number of the starting sector equals to $(m \cdot c_{\mathrm{size}}) + start$ like for RAID 0 configurations. The translation of a `TRIM` command in a RAID 5 is illustrated by an example shown in Fig. 4.

The main difference between RAID 5 and RAID 6 configurations is that the latter impose two parity chunks per stripe. The parity placement algorithms for RAID 5 configurations can also be applied to RAID 6 configurations. Therefore, we also use the left-symmetric parity placement for RAID 6 configurations in the following. For a RAID 6 configuration with n drives and $p = 2$ parity chunks per stripe, the stripe number m for a chunk range $(num : start : len)$ is given by $m = num \div (n - p)$. The index *p_drive* of the drive that holds the first parity chunk equates to $p_drive = (n - 1) - (m \mod n)$, while the index *q_drive* of the drive comprising the second parity chunk is computed as $q_drive = (p_drive + 1) \mod n$. The particular data chunk is located on the drive with the index $d_drive = (p_drive + (num \mod (n - p)) + 2) \mod n$. The number of the starting sector can be computed in the same way as for RAID 0

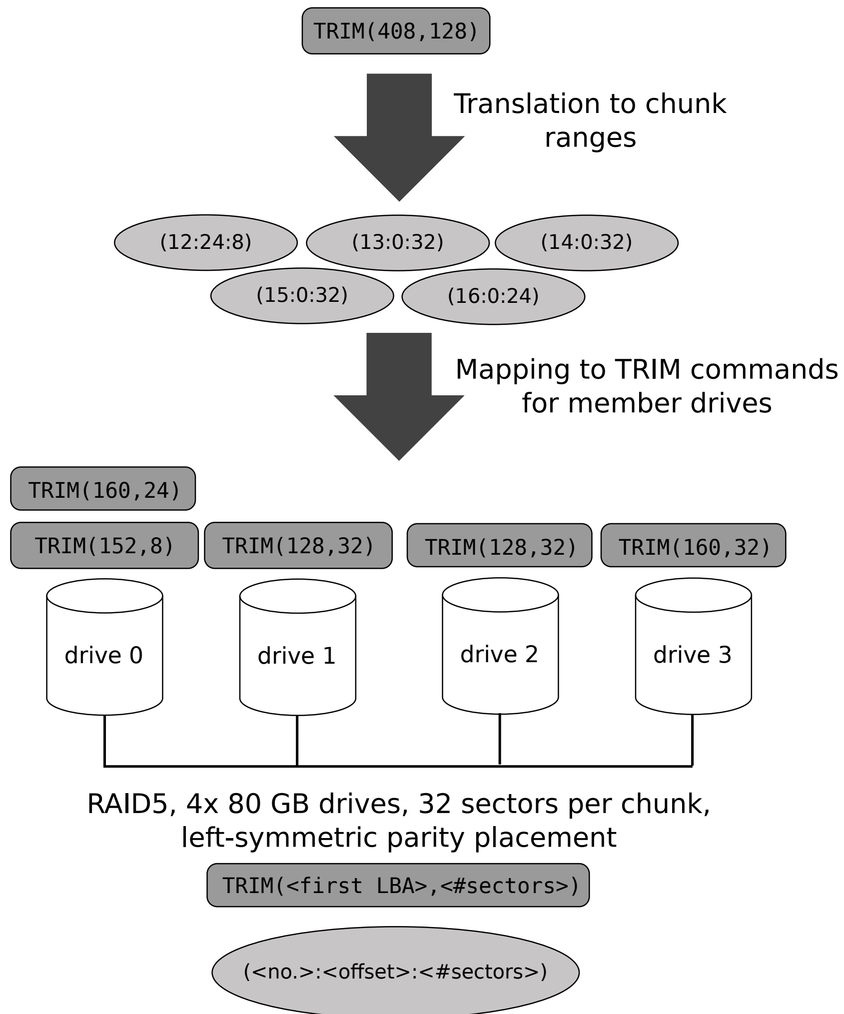


Fig. 4. Exemplary `TRIM` translation in a RAID 5 configuration with 4 drives.

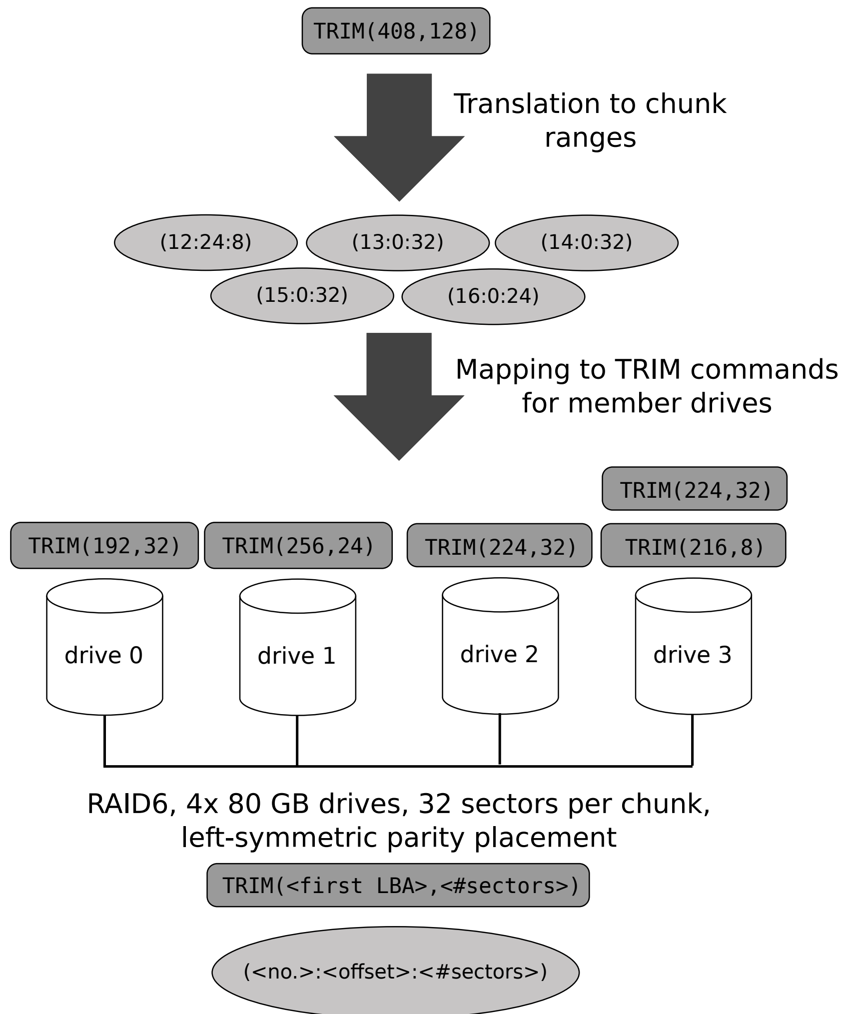


Fig. 5. Exemplary `TRIM` translation in a RAID 6 configuration with 4 drives.

and RAID 5 configurations. An exemplary translation process of a TRIM command in a RAID 6 organization is depicted in Fig. 5.

Since the parity information must always be consistent with the data, it has to be updated after a TRIM command was processed. A useful drive characteristic that would ease the maintenance of a consistent state in parity-based RAIDs would be that subsequent read requests to a trimmed LBA range always return the same data. However, the SATA standard [14] tolerates that subsequent reads of trimmed logical blocks may return different data. The exact behavior of a subsequent read request to a trimmed sector is reported by a particular SATA device when the *IDENTIFY DEVICE* command [14] is issued to it. There are three possibilities: The first is that each read of a trimmed sector may return different data, i.e., an SSD shows *non-deterministic trim behavior*. We denote this variant of trim behavior as "ND_TRIM" for the rest of this report. The remaining two possibilities represent a *deterministic trim behavior*, where all subsequent reads of a trimmed logical block return the same data, which can be either arbitrary (denoted as "DX_TRIM") or contain only zero-valued bytes (denoted as "DZ_TRIM"). The SATA standard [14] leaves for the variant DX_TRIM open whether the returned data will be the same for *different* sectors or not.

The variant DZ_TRIM permits the most comfortable handling of parity information updates. In this case, parity updates can be performed like for requests to write only zero-valued bytes into a particular logical block and without the necessity to read the trimmed logical block subsequently. A description of the parity update process for write requests in parity-based RAIDs can, e.g., be found in [4, section 24.3]. The reconstruction of data (e.g., in case of a drive failure or a erroneous sector) and the verification of data does not require particular attention if DZ_TRIM is used because all trimmed logical blocks will return only zero-valued bytes.

However, the situation is different for the remaining two cases: DX_TRIM and ND_TRIM. The variant DX_TRIM may require to read each of the trimmed sectors in order to recalculate the parity properly. This will entail additional read operations and further lower the throughput of the RAID device or at least extend the completion time of a TRIM command. The situation becomes even more complicated for ND_TRIM. The reason is that the stored parity information might be considered invalid during the reconstruction or verification of stored data because a read of trimmed sectors may deliver different values compared to the time of parity calculation. Both cases, DX_TRIM and ND_TRIM, can be handled by maintaining a permanently stored bitmap for all sectors in that the corresponding sectors are marked as either valid or trimmed. Then, the parity calculations are performed for a marked sector like for sectors that comprise only zero-valued bytes. Correspondingly, write requests with data solely composed of zero-valued bytes can also be superseded by a TRIM command instead of a writing operation. For the reconstruction or verification of data, the current values of the marked sectors are simply not considered. The described approach can also make reading trimmed sectors obsolete for DX_TRIM. Considering the impact of TRIM commands on the reliability of a RAID device, the situation is the same as for

regular write requests. If a failure (e.g., power loss, system crash) occurs during the writing of data, the parity information can become inconsistent with the data. However, for RAID setups maintaining a bitmap with the trimming state of each sector, this information may additionally become inconsistent in case of a failure.

In addition to trimming the LBA ranges covered by data chunks whose content is considered as no longer needed by a overlying filesystem, the storage space used by parity chunks can be trimmed as well in certain situations. This occurs when the resulting parity chunk would comprise only zero-valued bytes. Parity calculations are performed sector-wise, meaning that the first sector of a parity chunk contains the parity information generated by the first sector within each data chunk in a stripe. Due to this, it is possible that only a part of sectors within a parity chunk can be trimmed because they will contain solely zero-valued bytes. This method of trimming parity chunks in whole or in part can be applied to all three variants of trim behavior. However, the cases `DX_TRIM` and `ND_TRIM` require that trimmed sectors located within parity chunks are also marked within the bitmap used to maintain the trim status of each sector.

Hierarchical RAID Levels. Hierarchical RAID levels (e.g., RAID 10, RAID 55 and RAID 60) combine two RAID levels hierarchically by using a set of RAID devices as "member drives" for creating a new RAID device. For example, RAID 10 realizes a stripe set of mirrors, while RAID 60 comprises a stripe set of distributed double parity sets.

To support `TRIM` in a hierarchical RAID configuration, the translation of `TRIM` commands has to take place several times. Considering a RAID 60 organization, an address translation is first performed on the top-level RAID 0 and then for each of the underlying RAID 6 devices. In contrast to this, in a RAID 10 configuration, an address translation has to be performed only on the top-level RAID 0, while the resulting `TRIM` commands are simply duplicated and sent to each member drive of the bottom-level RAID 1 devices. An example of `TRIM` translation in a RAID 10 is depicted in Fig. 6.

Besides direct attached RAID devices, hierarchical RAID configurations can be found in *storage networks*, where a bunch of remote RAID devices is combined to a logical RAID device. For example, multiple RAID 6 devices deployed in different hosts can be connected through *Internet Small Computer System Interface (iSCSI)* and combined to one logical RAID 6 device.

3.2 Propagation of the `TRIM` command in RAIDs

Filesystems generate `TRIM` commands in order to notify SSDs of storage space parts with no longer needed content. These parts of storage capacity can be reclaimed by the garbage collection mechanism in order to gain more free flash capacity (cf. Sect. 2.2). Contemporary storage system architectures can include multiple intermediate layers between a filesystem and the physical block devices (e.g., SSDs). To ensure that the generated `TRIM` commands go down to the SSDs,

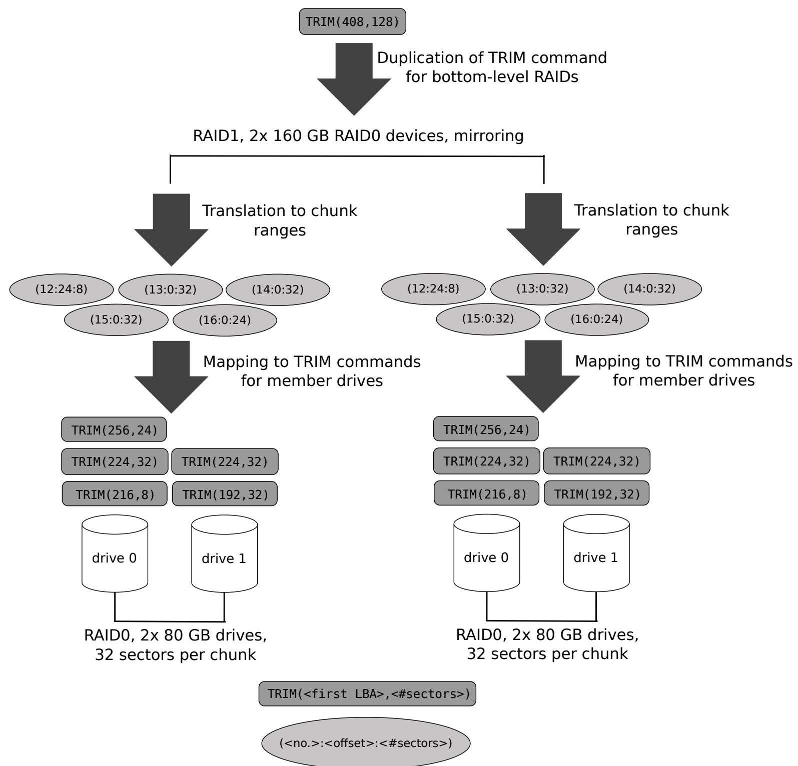


Fig. 6. Example of `TRIM` translation in a hierarchical RAID 10 configuration comprising 4 drives.

each intermediate layer must be able to propagate TRIM requests to the underlying layer. Our focus lies on configurations that employ RAID implementations as intermediate layers. Sect. 3.1 explains the steps required to appropriately handle TRIM commands at a RAID layer in order to propagate them to the to underlying layer. However, the performance penalty introduced by TRIM commands in combination with *NCQ* (Sect. 2.2) has to be taken into account.

It can be beneficial to delay the sending of TRIM commands to the physical block devices, e.g., when read and write requests using *NCQ* are processed or the utilization of a physical block device is high, in general. A further application case for delaying TRIM commands are situations, where it can be more advantageous to merge multiple TRIM requests in order to reduce their number and trim a larger part of storage capacity, for example, in workloads with many TRIM requests comprising only few sectors. The merging of TRIM requests is already implemented in some filesystems, e.g., *ext4fs* [9]. However, the problem is that intermediate layers often do not possess enough information about the current state of the physical block devices. Therefore, delaying TRIM commands should be performed at the layer that directly controls the physical block devices. Then, it would be sufficient when the intermediate layers would only pass the translated TRIM commands to the subjacent layer.

An unfavorable side effect of delaying TRIM commands that has to be considered is a possible inconsistency in parity-based RAIDs when TRIM requests are delayed too long. The reason is that a device failure can prevent the processing of the delayed TRIM requests although the parity information was already updated considering the trimmed sectors. Furthermore, a race condition between TRIM commands and subsequent read requests can occur. Both problems can be avoided by overwriting the sectors addressed by the present TRIM requests for example with zero-valued bytes, if the physical block device exhibits the variant DZ_TRIM of trim behavior. In the cases DX_TRIM and ND_TRIM, overwriting sectors referred to by outstanding TRIM commands is unnecessary due to the fact that they are marked as trimmed and, hence, treated like containing solely zero-valued bytes. Another option for avoiding inconsistencies due to delayed TRIM is keeping track of outstanding TRIM requests using a bitmap, which is stored permanently and can survive a failure like a power loss. Further issues can occur when TRIM requests are delayed for varying lengths of time on the different physical block devices. For such cases, there should be a possibility of forcing immediate processing of outstanding TRIM commands on all physical block devices.

4 Conclusions and Future Work

Incorporating the TRIM command into SSD-based RAID architectures is an important step to maintain high transfer and I/O rates for write requests in such architectures. This task can be more conveniently achieved for parity-less RAID configurations. However, incorporating TRIM into a parity-based RAID setup results in new challenges mainly resulting from the read behavior of trimmed

sectors. Furthermore, it can result in overhead caused by additional parity calculations and drive accesses. Nevertheless, the incorporation of the `TRIM` command into RAID configurations promises an increased RAID write performance when the interaction between `TRIM` commands and *NCQ* was appropriately taken into account.

Our next step will be the development of a prototype implementation of a `TRIM`-aware parity-based RAID. Based on this prototype, we will be able to study the behavior of a `TRIM`-aware parity-based RAID configurations. We will focus on the quantification of the additional overhead introduced by the required parity update operations. By this, we expect to get insights into how to build efficient high-performance SSD RAID systems.

A further direction would be to develop an architecture supporting the propagation of `TRIM` not only in RAID implementations but also in all intermediate indirection layers between the filesystem and the SSDs. In such architectures each indirection layer has to translate `TRIM` commands and pass them directly to the underlying layer, except for the lowest layer that owns the exclusive control over the physical block devices and, hence, will have an reliable utilization information. This will allow to realize an efficient delaying and merging of `TRIM` requests intended for a particular physical block device.

References

1. Burkhard, W., Menon, J.: Disk array storage system reliability. In: Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on. pp. 432 –441 (jun 1993)
2. Chen, P.M., Lee, E.K., Gibson, G.A., Katz, R.H., Patterson, D.A.: Raid: high-performance, reliable secondary storage. ACM Comput. Surv. 26(2), 145–185 (1994)
3. Hu, X.Y., Eleftheriou, E., Haas, R., Iliadis, I., Pletka, R.: Write amplification analysis in flash-based solid state drives. In: SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference. pp. 1–9. ACM, New York, NY, USA (2009)
4. Jacob, B.L., Ng, S.W., Wang, D.T.: Memory Systems: Cache, DRAM, Disk. Morgan Kaufmann (2008)
5. Lee, E.K., Katz, R.H.: Performance consequences of parity placement in disk arrays. In: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. pp. 190–199. ASPLOS-IV, ACM, New York, NY, USA (1991), http://doi.acm.org/10.1145/106972.106992
6. Lee, E.K., Katz, R.H.: An analytic performance model of disk arrays. In: Proceedings of the 1993 ACM SIGMETRICS conference on Measurement and modeling of computer systems. pp. 98–109. SIGMETRICS '93, ACM, New York, NY, USA (1993), http://doi.acm.org/10.1145/166955.166994
7. Lee, E.K., Katz, R.H.: The performance of parity placements in disk arrays. IEEE Transactions on Computers 42, 651–664 (1993)
8. Leventhal, A.: Triple-parity raid and beyond. Commun. ACM 53(1), 58–63 (2010)
9. Mathur, A., Cao, M., Bhattacharya, S., Dilger, A., Tomas, A., Vivier, L.: The new ext4 filesystem: current status and future plans. Linux Symposium (2007), http://ols.108.redhat.com/2007/Reprints/mathur-Reprint.pdf

10. Menon, J., Riegel, J., Wyllie, J.: Algorithms for software and low-cost hardware raids. In: Proceedings of the 40th IEEE Computer Society International Conference. pp. 411–. COMPCON '95, IEEE Computer Society, Washington, DC, USA (1995), http://dl.acm.org/citation.cfm?id=527213.793536
11. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (raid). SIGMOD Rec. 17, 109–116 (June 1988)
12. Serial ATA International Organization: Serial ATA Revision 2.6. Tech. Rep. Revision 2.6, Serial ATA International Organization (February 2007)
13. Shu, F., Obr, N.: Data Set Management Commands Proposal for ATA8-ACS2. Microsoft Corporation, One Microsoft Way, Redmond, WA. 98052-6399, USA, revision 1 edn. (Jul 2007)
14. Technical Committee T13: Information technology – ATA/ATAPI Command Set - 2 (ACS-2). Tech. Rep. T13/2015-D, revision 7, Technical Committee T13 (June 2011)