Adapting the Data Organization of Secondary Storage in Virtualized Environments

Nikolaus Jeremic[1], Helge Parzyjegla[1], Gero Mühl[1], and Jan Richling[2]

[1]*Architecture of Application Systems Group, University of Rostock, Germany*
{nikolaus.jeremic,helge.parzyjegla,gero.muehl}@uni-rostock.de

[2]*Communication and Operating Systems Group, TU Berlin, Germany*
jan.richling@tu-berlin.de

Abstract: In virtualized environments, multiple virtual machines (VMs) usually share a common secondary storage system which is, thus, often subject to a broad range of access patterns and different requirements (e.g., regarding performance, capacity, and reliability) imposed by diverse applications running inside the VMs. Moreover, with applications and VMs being added, started, stopped, and removed, access patterns as well as requirements may vary significantly over time. Ideally, the storage system is able to adapt to changing access patterns while considering application requirements at the same time. However, many storage systems only use a static data organization scheme in terms of drive assignment and data layout that is defined at deployment time, and may become disadvantageous or even inappropriate for the current workload. Although some storage systems employ data migration (e.g., dynamic storage tiering), the data layout remains unchanged due to a prohibitive high reorganization overhead.

In this paper, we propose a mechanism for a fine-grained data organization adaptation that includes the data layout. This significantly extends the range of feasible adaptions compared to existing systems. Our approach factors application hints and requirements into adaptation decisions and exploits observations of access patterns as well as the state of the page cache to increase its effectiveness. Furthermore, we present a case study showing the benefits of fine-grained adaptations and discuss two options for the integration of the proposed adaptation mechanism into existing virtual machine monitors (VMMs), also known as hypervisors (HVs).

1 Introduction

For many data-intensive applications, I/O performance is critical. This is especially true for those applications designed to produce, store, access, and analyze large amounts of data. Ensuring the necessary I/O performance for such applications in virtualized environments is a challenging task. On the one hand, data layout, storage infrastructure, hypervisor, (guest) operating system (OS), and applications need to be aligned and fine-tuned with each other in order to optimize their performance, while, on the other hand, virtualized environments usually aim to hide details of the actual physical implementation in order to ease the management of hosted virtual machines (VMs). The latter, thus, makes it not only difficult to align storage infrastructure and applications, but also leads to many indirections within the storage stack causing a considerable performance overhead.

Moreover, with applications and hosted VMs being dynamically started, stopped, added, and removed, the overall access pattern on a shared secondary storage system may change substantially over time. Hence, an initial data organization scheme that was once optimized for a given set of applications and VMs may not be suitable anymore or may even be inappropriate for the present workload. For instance, a hybrid storage system may keep frequently accessed data items on fast solid-state drives (SSDs), while the majority of data items that are accessed less frequently are stored on much slower, but larger hard disk drives (HDDs). The more the access pattern changes, the less advantageous the initial data distribution may become, leading to a decreasing I/O performance. Furthermore, the RAID policy [PGK88] by which the storage system combines drives to a storage array already implies a particular trade-off between performance (for specific access types), reliability, and net storage capacity (i.e., the usable fraction of the gross storage capacity of the involved drives). New VMs and applications being added to a virtualized environment pose different requirements to the storage system (e.g., regarding reliability or storage capacity), potentially making the once chosen RAID policy inadequate.

Therefore, it is necessary to adapt the data organization to new requirements and changing access patterns. Dynamic Storage Tiering (DST) [CKZ11, GPG$^+$11, AvMT12] can be applied to automatically determine, manage, and update the set of hot data items that are accessed frequently and, thus, should be stored on faster drives. However, the effectiveness is often limited by the coarse granularity of performed adaptations that usually consider only large data items or whole volumes. Such adaptations reflect occurring access patterns only insufficiently and also cause a considerable reorganization overhead. Additionally, the restricted visibility of the overall system state (e.g., cached data items) due to several indirections within virtualized environments also prevents more efficient implementations. To overcome these limitations, we aim at integrating an adaptive storage logic within a storage stack designed for virtualized environments. This allows for the implementation of both efficient and fine-grained adaptations of the data organization as well as the layout on involved drives. In particular, we make the following contributions:

1. We propose a fine-grained scheme for adapting the data organization that allows to capture access pattern changes more accurately and minimizes the costs for reorganization due to a smaller reorganization scope.
2. We describe a control mechanism for the above scheme with an extended view of the memory system state (including page cache) that also considers explicitly expressed applications hints and requirements besides observations of data access patterns.
3. We present a exemplary instance of the proposed adaptation mechanism based on adaptive RAID level reconfiguration along with an evaluation based on simulation.
4. We discuss options for the integration of the proposed adaptation mechanism into existing virtualized environments.

The remainder of this paper is organized as follows: Background information on data organization, RAID policies, and storage in virtualized environments is provided in Sect. 2. Sect. 3 gives a general overview about our approach, while Sects. 4 and 5 elaborate on details of the adaptation logic, and discuss several integration alternatives, respectively. In Sect. 6, we finally conclude the paper by summing up its main contributions and presenting further research directions.

2 Data Organization, RAID Policies, and Storage Stacks

This section provides the background on data organization in secondary storage systems in general and RAID (Redundant Array of Independent Disks) policies in particular as well as their implementations in storage stacks for virtualized environments.

Data Organization. Advanced secondary storage systems comprise multiple drives, potentially of different types (e.g., HDDs and SSDs) and, thus, with different characteristics. The *data organization* reflects how a dynamic set of data items is stored on the available drives. The mapping of data items to drives is defined by a *data organization policy* which has a major influence on the *reliability* (with respect to a particular fault model), the *storage space requirements* and the *performance* (with regard to bandwidth and latency of read and write requests). A particular data organization policy comprises information on applied organizational concepts (e.g., data mirroring and data striping potentially with erasure coding), involved drives, and concept-specific configuration parameters (e.g., number of data copies or parities, stripe unit/chunk size) as well as their implementation. Hence, the data organization policy defines on which drives what data item or which parts of it are stored and how they are laid out on the individual drives.

In most cases, data is organized in *files* that are grouped into *directories* which itself are implemented as special files in many file systems. Thus, the task is to map files (and their metadata for bookkeeping) to *blocks* that correspond to portions of the storage space of involved drives. The implementation of a concrete data organization policy provides such a mapping. It can take place in several stages and involve multiple layers, for instance, if *RAID layers* or *logical volume managers* are placed as intermediate indirection layers between the file system and the physical storage devices in order to aggregate the storage capacity of multiple drives. However, RAID and logical volume management can also be combined and incorporated into the file system, like in the case of *Btrfs* and *ZFS*. Due to the possibility to map files to blocks in multiple stages, different *types of mapping* are distinguished: block-to-block (e.g., logical volume managers, block-level RAID), block-to-file (e.g., virtual disk backed by files), file-to-block and file-to-file (e.g., stacked file systems).

RAID. RAID [PGK88, KGP89] is a data organization technique that relies on *data mirroring* and *data striping* potentially with *parity* in order to combine multiple drives into a single logical drive. A certain RAID setup is specified by a *RAID policy* which is an instance of a data organization policy. The RAID policy comprises information about the RAID level determining the employed organizational concepts, the identity of the used drives and further configuration parameters that are specific to the chosen RAID level and its implementation (e.g., data/parity layout or mirroring scheme). A RAID can be implemented in hardware by a RAID controller or in software by the OS.

The RAID fault model assumes the failure of whole drives, hence, reliability is measured by the maximum number of drives that may fail simultaneously without losing data. In order to retain data in the event of drive failures, the data is either mirrored (RAID 1) or supplemented with parity information (RAID 2 to 6) which enables the reconstruction of the data on failed drives. The number of tolerable simultaneous drive failures is then determined by the number of copies and parities, respectively. The net storage capacity of a RAID results from the gross storage capacity of the available drives as well as the number of copies and employed parities. The performance of a particular RAID setup depends on the request pattern, the RAID policy, and its implementation. Thereby, each RAID policy provides a certain trade-off between usable storage capacity, available performance and ensured fault tolerance. Based on the number of drives, policy parameters can be varied to setup a RAID configuration fulfilling given capacity, performance, and reliability requirements. However, an improvement regarding to one of the characteristics goes along with a degradation in at least one of the others. Thus, no RAID policy is optimal with respect to all three characteristics.

Storage Stack in Virtualized Environments. In a virtualized environment, several VMs (called *guests*) are run on a physical machine (called *host*) whose resources are virtualized and managed by a software layer referred to as *hypervisor* (HV) or *virtual machine monitor* (VMM)[1]. The most common method to provide secondary storage in such environments is to expose virtual disks to guests through a block-based standard disk interface using the ATA or SCSI command set. This enables the guest to create an arbitrary file system on a virtual disk in order to organize its data or to use several virtual disks with an indirection layer (e.g., a logical volume manager). Virtual disks are emulated or paravirtualized. Their storage capacity is backed by files denoted as *virtual disk image (VDI)*, by local block devices or by remote block devices connected via a *storage area network (SAN)*, e.g., using protocols like iSCSI or SRP. In the case of file-backed virtual disks, the HV either organizes VDIs in a special file system on top of local or remote block devices, or it uses a file-based *network-attached storage (NAS)*, e.g., using the NFS or CIFS protocol. Block devices used to back a virtual disk can be physical storage devices (e.g., a hardware RAID) or partitions on such devices, as well as logical block devices resulting from interposed indirection layers (e.g., software RAID and/or a logical volume manager).

As an alternative to virtual disks, storage adapters of the host can be passed through to a VM. In this case, the VM has direct and exclusive access to the attached devices. Furthermore, a VM can also directly import block devices from a remote storage server or mount a remote file system. Another possibility is to have the HV pass through a paravirtualized file system to the guests [JVHLP10].

3 Adaptive Data Organization in Virtualized Environments

In virtualized environments, there is always the trade-off between isolating the VMs from the host and maximizing their efficiency. As introduced in the previous section, the storage stack in virtualized environments contains several layers both inside the VM and inside the HV (or even in remote storage servers). Although this layered architecture offers a clean separation of concerns regarding functionality, it leads to numerous indirections

[1]Please note that we do not consider container-based virtualization here due to its significant disadvantages regarding the independence of host and guest. However, in such environments our approach is easily applicable because of the quite simple storage stack allowing guest applications to directly access the host's file system.

deteriorating the I/O performance. In particular, these indirections restrict the visibility of meta information required for further optimizations such as an automated adaptation of the data organization. In order to effectively adapt the data organization as much meta information as possible is required that, for instance, includes the mapping of files to blocks or the access patterns of files in relation to the directory they belong to. For a more holistic approach, this information needs to be passed and shared between the individual layers of the storage stack.

In a virtualized environment, this means that information usually hidden inside the VM must be made available to the HV. For example, file system related information from inside the VM can be passed to the host system by means of a paravirtualized file system or by enriching existing block-based interfaces to pass meta information gathered in the VM to the HV. Another possibility is to merge adjacent layers, e.g., RAID and volume manager. This increases the possible range of optimizations and extends the amount of available meta information for decision-making. This is, however, only possible for layers that are either located all in the VM or all within the host.

In the following, we discuss our approach in a bottom-up manner. We start with describing a generic method for an automated adaptation of the data organization in Sect. 4. Then, we discuss in Sect. 5 two alternative options how the adaptation mechanism can be integrated into a storage stack for virtualized environments.

4 Automated Adaptation of the Data Organization

An adaptation of the data organization is beneficial in several situations. As the performance of a particular data organization is sensitive to the access pattern, the performance may deteriorate if the pattern changes. Additionally, storage devices are not equally suited for all request patterns, e.g., random requests with small size are slow on HDDs while SSDs are usually much faster but can exhibit a low write performance when lacking spare capacity. To counter this, it is advantageous to migrate data to devices that are better suited for the present access pattern or to reorganize the data in order to increase the spare capacity on SSDs. Besides the access pattern, performance and reliability requirements of applications may also change over time. As a result, it may be necessary to reorganize the corresponding data to meet these new requirements. However, these kind of adaptions should be automated in order to react quickly on short-term changes as well as to unburden system administrators from these complex and error-prone tasks.

There are remarkable efforts to design self-tuning storage systems. One direction is to adapt the drive allocation of data items based on their access frequency and pattern by migrating them to drives that are generally faster or better suited for the respective access pattern. This is the basic concept of *Dynamic Storage Tiering* (DST), where drives of different types such as HDDs and SSDs in hybrid storage systems [WR09] or alternatively HDD RAIDs with different configurations [WGSS96] are considered as separate storage tiers that represent different levels in the computer memory hierarchy. Another direction is to automatically tune configuration parameters of the organizational concept to the workload, for instance, by adapting RAID parameters such as stripe unit (chunk) size and stripe width [SWZ98]. For both approaches, it is beneficial to specify a fine-grained data organization scheme. However, this is often not viable because storage capacity is usually partitioned into (logical) volumes comprising multiple gigabytes which have to be reorganized at once causing a prohibitive high reorganization overhead. Even if the RAID policy is specified on a per-file or per-directory basis [AvMT10] the reorganization overhead may still be considerable due to directories with many and/or large files.

Moreover, technology-specific characteristics of the storage devices are often addressed only insufficiently by existing approaches. For SSDs, for example, it is usually not considered that the write performance, especially for small random writes, significantly depends on the current amount of available spare capacity [JMBR11]. Consequently, SSDs with different levels of space utilization and, therefore, different amounts of spare capacity can be considered as different *storage tiers* with regard to small random writes. A further limitation is that reorganization decisions rely solely on the observation of access frequency and patterns and may, depending on the observation period, sometimes be misleading. In these cases, the usage of additional access hints [BN12, YVZS09] may lead to much better placement decisions. Furthermore, when reorganizing the data layout, it is usually not checked whether affected data items are already present in the the main memory page cache that, when exploited, can reduce the reorganization overhead significantly. In fact, each of the aforementioned approaches aims at improving the I/O performance, but the separation of layers within the storage stack prevents their alignment and orchestration in order to leverage their combined potential. In the following, we thus pursue a more holistic approach to adaptive and self-tuning heterogeneous secondary storage systems.

4.1 Data Organization Scheme

The data organization scheme described in the following is based on the notion of files that are partitioned into smaller *file segments* with each file segment being subject to an own data organization policy (e.g., RAID policy). Hence, two segments of the same file may have a different layout. Depending on the requirements with regard to performance, reliability and storage space, a file segment may get replicated (mirrored) on multiple drives, striped over multiple drives with or without parities, or it can reside on only one of the available drives. If a file segment is spread across multiple drives, disjoint drives are chosen at random in order to distribute larger requests over several drives (if striping is applied) and balance the load. If striping is applied for a particular file segment, it is spread over multiple chunks (stripe width is specified by the data organization policy) with equal maximum length. These chunks are allocated to drives independently of their position within the file segment and the mapping of each chunk to a particular drive is stored in the file system's metadata. This flexible allocation scheme avoids the relocation of a chunk in certain reorganization cases, e.g., when adding or removing parities in connection with a corresponding adjustment of the stripe width, i.e., the number of chunks in a file segment.

However, a potential issue is fragmentation due to growth/truncation of a file or due to relocation of data in a file segment as a consequence of reorganization. In the case of SSDs, this is not a real issue since SSDs have no moving parts and, thus, do not suffer from rotational or head positioning delays like HDDs. Furthermore, logical address ranges are not necessarily mapped to a contiguous storage region due to out-of-place updates. As a result, fragmentation is only an issue in the case of HDDs. In order to work against fragmentation, allocation techniques like extent-based delayed allocation [SDH$^+$96], which were developed to tackle this issue, can be employed.

Another issue is a potentially large amount of metadata due to fine-grained data organization policies in connection with large files that contain many file segments. However, this can be mitigated by increasing the size of file segments and, thus, reducing the amount of metadata that has to be maintained and stored. As an example, when a large file is created, it is possible to make file segments contain several megabytes of data. Later, big file segments are split into multiple smaller file segments when necessary. In particular, this gives the opportunity to refine the data organization policy only where it is beneficial.

Please note that the proposed approach, with some restrictions, is also applicable on block level. Several blocks with similar access characteristics are then considered as a block segment that becomes subject to similar reorganizations as described above. However, a the information about the mapping between files and block segments is lost in this case.

4.2 Automated Data Reorganization

With our approach, the data organization policy is specified on the basis of file segments. Hence, in the course of reorganization, file segments are migrated between drives and/or rearranged by splitting or merging them. In the following, we discuss how decisions are taken, which strategies are followed, and what reorganization overhead is caused thereby.

Basis of Decision Making. Reorganization decisions are based on workload observations, application requirements and access hints as well as the current state of the page cache and the incurring reorganization costs.

Workload Observation. The workload is observed by capturing the characteristics of requests to a file segment. This includes the access frequency as well as the access pattern comprising the request size and type (i.e., read or write). In this way, frequently accessed file segments can be kept on SSDs, while others are stored on HDDs. Furthermore, file segments that are mainly read may be stored on SSDs with a high storage capacity utilization without suffering from decreased write speed due to a smaller amount of spare capacity. The information about the prevalent request size can be exploited to adapt the chunk size and stripe width. Furthermore, access frequency and request size allow to identify performance-critical data [CKZ11] and help to detect if a certain data item is accessed rather in a random (small requests) or in a sequential (larger requests) manner.

Application Requirements. Requirements can refer to a certain level of performance and reliability. In order to prioritize the requests to the data of particular applications, performance classes can be defined. When new data is created, the performance class can be specified on a per-directory or per-file basis. Similar to this, a reliability level can be specified in terms of the maximum number of simultaneous drive failures without losing the content of a particular file or all files in a certain directory.

Page Cache State. The information about cached file data can be exploited in order to take less read requests into account for the calculation of reorganization costs. However, the cached file data is evicted from main memory in the short term. Thus, it is advantageous to reference the cached file data during the decision process in order to postpone its eviction and reduce the likelihood that it has to be read from the much slower drives if the decision was taken for a reorganization. In virtualized environments, however, special care must be taken because data might be cached by the guest and also by the host OS potentially leading to an inefficient operation. To solve this problem, appropriate interfaces and/or a coordinated control of caching for both layers is required.

Reorganization Overhead. An important factor affecting reorganization decisions is the incurred overhead to reorganize a file segment in terms of read and write requests. The number of the imposed requests depends on the current data organization of the file segments in question as well as the targeted data organization to be established. In the worst case, the whole data of a file segment has to be relocated and/or replicated to different drives. However, in many cases reorganizations can be executed with much less overhead.

Access Hints. Access hints provide information on the expected future use of data. Such hints comprise information on a variety of access characteristics: prevalent access type (read or write), manner (random or sequential), frequency and properties like request size. Moreover, hints can be given on the intended use of data, for example, that data is only temporary. Hints on future data access are useful in two respects: First, they can be used to decide on an appropriate initial organization. Second, access frequency and pattern can change over time, and a reorganization can take place in a more focused way when hints about the future use of data are available.

Storage Capacity Utilization. The storage capacity utilization can have a large impact on performance when SSDs are used. This can be exploited to establish additional SSD storage tiers by maintaining an uneven capacity utilization of SSDs. For example, the storage capacity of some SSDs can be filled to a larger extent with data that is mostly read because the read speed does not suffer from low spare capacity like the write speed. In contrast to this, frequently written data should be stored on SSDs with lower storage capacity utilization in order to maintain a higher write speed.

Strategies. Changes of the workload and of application requirements trigger different data reorganization actions depending on the applied strategies. In the following, we describe strategies for the considered subjects of changes.

In reaction to changes of the access frequency, file segments are migrated to a different storage tier. In case of an increase in access frequency, the file segment is migrated to a higher and faster storage tier. If it is already located on the top tier, the organizational concept is changed or its configuration is adapted in order to increase the performance under the prevalent access pattern. In case of a decrease in access frequency, the file segment is migrated to a lower and slower storage tier. Once it reaches the lowest storage tier, its data organization is changed to reduce the required storage capacity. If the access pattern changes, as a first measure, the organizational concept can be changed in order to better fit to the new pattern. However, if the currently employed organizational concept is generally suitable, parameters may be adapted, for instance, if the request size changes and data striping is used, the chunk size and stripe width can be adjusted.

Changes of the reliability requirements mean that the number of tolerable simultaneous drive failures should be either decreased or increased. If data mirroring is applied, the number of copies is correspondingly decreased or increased, respectively. If data striping is applied, this can be achieved by removing or adding parity. However, when changing from no redundancy to some redundancy level, a technique is chosen depending on the access frequency and the access pattern. For less frequently accessed data or mostly read data, striping with parity is favorable due to lower storage space demands and potentially (depending on configuration and request size) high read performance. For data that is frequently subject to small (random) writes, mirroring is more advantageous due to the write penalty of striping with parity. The storage tier is chosen on the basis of the access frequency, desired performance level, and the current space utilization. If data mirroring is used, copies can be placed in different storage tiers (e.g., one copy on SSD and two copies on HDDs) in order to keep enough free space in higher storage tiers, but still increase the reliability.

In reaction to changes regarding performance requirements, depending on the situation, it can be beneficial to migrate file segments between storage tiers, to switch the whole organizational concept, to just adapt its configuration parameters, or to perform a combination of these measures.

Reorganization Control. The reorganization is guided by a rule-based reorganization policy. Rules are conditional expressions over variables containing values of system parameters and thresholds, simultaneously reflecting optimization constraints imposed by the defined requirements and objectives as well as priorities between these according to the underlying strategies. In order to determine the most appropriate data organization for a file segment, a generic bonus-malus system is employed. As a result, an assessment of the current data organization is made in the event of a request to a file segment. If the data organization is suitable for the current request, a bonus is granted in order to encourage the actual data organization. Otherwise, a malus is taken into account. The ratio between bonus and malus is represented by a score. To avoid that too much bonus is accumulated (impairing the reactivity), the bonus is cut off when the score reaches a predefined lower limit. If the score reaches a predefined threshold (upper limit), reorganization of the corresponding file stripe is triggered. This takes place always after serving the particular request to keep its latency low. The threshold depends on the reorganization overhead and is adapted to counter oscillations.

Please note that reorganizations are only triggered when data items are accessed and their score gets updated. Hence, rarely accessed items or those that are not accessed at all elude from reorganization and may waste valuable capacity on faster storage tiers. To tackle this, a malus is periodically added to all items of a storage tier, so that only accessed frequently items will accumulate enough bonus for compensation in the meantime. Thus, rarely accessed items will eventually be migrated to lower tiers or reorganized into a more space-efficient layout.

Reorganization Overhead. The overhead of a particular data reorganization depends on the individual reorganization steps and actions, the presence of affected data in the page cache, and potentially on the actual utilization of the storage capacity on the involved drives. Any actions taken to reorganize data impose further read and write requests. Thus, the reorganization overhead is measured in terms of incurred requests. However, read requests can be served without accessing drives if the data is stored in the page cache, hence, reducing the reorganization overhead described in the following.

Switching from striping to mirroring requires to replicate each chunk of a file segment in order to produce a certain number of copies. Thus, each chunk is first read and then written to at least one other drive. In the opposite case, a mirrored file stripe is read and spread over multiple chunks that are written to different drives. If parity is also employed, all parities must be calculated and written. Adding or removing a parity chunk to/from a striped file segment is least costly when combined with a corresponding increase or decrease of the stripe width. In such cases, adding a parity chunk requires to read all data chunks, calculate the parity and to write the new parity chunk to one of the remaining drives. If a parity chunk has to be removed, no further action is necessary besides a metadata update for the file segment in question. If the stripe width has to be preserved, the chunk size can be adapted imposing at least partial relocation of data chunks. Preserving the chunk size and stripe width is least favorable, because it potentially requires to reorganize multiple file segments. These considerations also apply if only the stripe width or the chunk size are changed, respectively. Increasing the number of copies of a mirrored file segment requires reading the whole file segment and writing its data to one of the remaining drives. Decreasing the number of copies requires no further data access besides a metadata update. Moreover, switching between different mirroring schemes imposes at least a partial relocation of the data contained in a file segment.

4.3 Case Study: RAID Level Reconfiguration

In this section, we describe an adaptive RAID level reconfiguration mechanism as an exemplary instance of the proposed approach. The goal is to maintain high write throughput by adapting the RAID level to the request size. As we demand that no data is lost when any two drives fail simultaneously, we, thus, consider switching between RAID 6 and RAID 10 with triple mirroring. Furthermore, we use file segments of equal (maximum) size that are spread over 6 drives (minimum number of drives). As a result, on reconfigurations, either a RAID 6 stripe is split into 2 RAID 10 stripes or 2 adjacent RAID 10 stripes are merged into one RAID 6 stripe without the necessity to relocate any data chunk.

RAID Level Reconfiguration Policy. Please note that the write throughput depends on the request size that determines the number of affected data chunks within a file segment (up to 4). In particular, each number of affected chunks imposes a different number of read and write requests. The following analysis is based on the (simplifying) assumption that reading and writing a particular block takes the same time. Thus, the write overhead is expressed as the number of I/O requests (neglecting potential I/O merging on the block

Affected Chunks	RAID 6	RAID 10	Difference in #I/Os
1	3R + 3W	**3W**	3
2	2R + 4W	6W	0
3	**1R + 5W**	9W	3
4	**6W**	12W	6

Table 1: Overview of write overhead for RAID 6 and RAID 10 (lower overhead is marked in bold).

layer). Consequently, the comparison of the write overhead is solely based on the total number of imposed I/Os as summarized in Table 1. For example, according to the first line, a write affecting one chunk incurs 6 I/Os for RAID 6 but only 3 I/Os in the case of RAID 10, saving 3 I/Os each time. Taken together, if the current RAID level is better suited for the number of chunks affected by a write request, 3 or 6 I/Os can be saved. Based on this analysis, we employ the following bonus-malus-system: For each file segment a non-negative score is maintained, representing the number of additional I/Os due to the less suited RAID level. If the actual RAID level is better suited, the score is decreased by the number of saved I/Os, but cut off at 0. Each file segment is initially stored in the RAID 6 layout and the values of the reconfiguration thresholds are given by the reconfiguration overhead. For switching from RAID 6 to RAID 10, the threshold is set to 12 (4 chunks are read and each is then written twice more). In the opposite case, the threshold is set to 6 (4 chunks are read for parity calculation and 2 parity chunks are written).

Simulation. The described heuristic is compared to an offline algorithm as well as to static RAID 6 and RAID 10 setups by simulation. The offline algorithm determines for each file segment which initial RAID level and what reconfigurations will minimize the total number of I/Os based on the knowledge of all performed (future) write requests. Thus, its result is optimal with respect to the analytical model above where the write throughput is the higher the less subsequent I/O operations are performed. The considered workloads represent overwriting data in a set of files with different request sizes and, hence, affecting different numbers of chunks in a file segment. The accessed file segments are chosen randomly. We consider 5 different workloads: For workload A the size of each request is chosen randomly making each number of affected chunks (almost) equally likely. In contrast to this, workloads B_1, B_2, C_1 and C_2 contain a number of recurring write requests for the same file segment that last for a certain amount of time (for B_2 and C_2 four times longer than for B_1 and C_1). Moreover, B_1 and B_2 favor larger requests (3 and 4 chunks), while C_1 and C_2 are biased to smaller request sizes (1 chunk). The simulation results are depicted in Fig. 1 showing the relative number of necessary I/O operations compared to the optimal strategy (smaller is better). If requests are performed purely random as in workload A, there is no access pattern to be exploited. Thus, the optimal strategy is to stick to the space-efficient RAID 6 layout that is superior or at least equal in the majority of cases (i.e., for 2, 3, and 4 chunks). Our heuristic imposes a negligible small overhead due as it is quite unlikely, but not impossible, to reach the threshold triggering an unnecessary reconfiguration in this case. Looking at the other workloads, the situation is different as an access patterns lasts long enough to adapt to it. Please note that neither the static RAID 6 nor RAID 10 layout are optimal anymore since each of these workloads contains

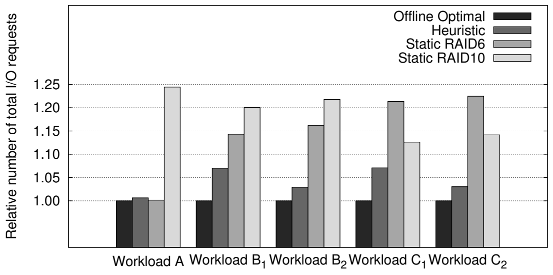
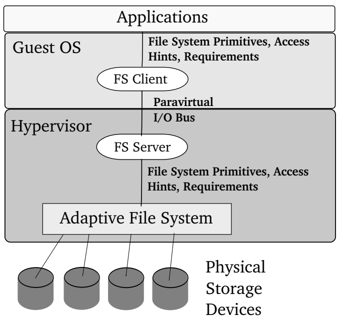


Figure 1: Comparison of RAID level adaptation to static RAID level setups with regard to the number of performed I/O requests under different workloads.

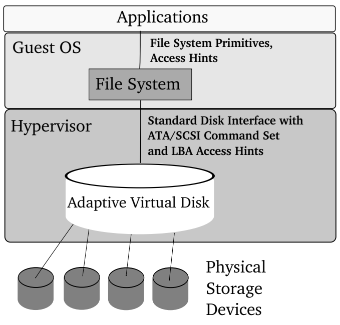
a differently biased mix of smaller and larger requests. Compared to the optimal offline algorithm, the results show how many I/Os can be saved by a fine-grained adaptation of the data layout. The overhead of our heuristic is mainly caused by threshold that has to be reached first before the access pattern is recognized and a corresponding reconfiguration is triggered. Nevertheless, the heuristic achieves better results than both static setups, paying the more off the longer a specific pattern lasts (comparing B_1 to B_2 and C_1 to C_2).

5 Integration into Virtualized Environments

The approach introduced in Sect. 4 relies on the availability of information regarding file accesses issued by the application. Usually, this information is available to the file system and, therefore, present in the OS. In virtualized environments with guest and host OS, this is not the case as several indirection layers usually restrict the information flow to the required minimum and, thus, hide details such as file access pattern. Therefore, the goal is to adjust the storage stack of virtualized environments in a way that as much information as possible can flow from applications inside a VM issuing the file access down to the drivers that actually deal with physical devices. In order to achieve that, it is ideal if the secondary storage is presented by the HV to the guests through a paravirtualized file system passing file accesses from within the guest to a storage facility in the host. If this is not possible (e.g., due to compatibility reasons), the secondary storage has to be presented to the guests as a block device, which can be either emulated or paravirtualized. In both cases, the useed interface should be enriched such that requirements and hints are passed to the HV, too. On the host layer, the mechanism described in Sect. 4 can be implemented either as a file system or a logical block device. Moreover, the block device can be local (exported by the HV) or on a remote machine (exported by a storage server). This offers a variety of combinations. However, we focus on combinations where the type of interface between guest and HV matches the abstraction level of the implementation of the proposed mechanism: A paravirtualized adaptive file system and an emulated or paravirtualized adaptive block device. The remaining combinations can be realized similarly by employing an indirection layer that performs an appropriate mapping in order to glue interfaces of different types.



(a) Paravirtualized adaptive FS.


Applications
Guest OS
File System Primitives, Access Hints
File System
Hypervisor
Standard Disk Interface with ATA/SCSI Command Set and LBA Access Hints
Adaptive Virtual Disk
Physical Storage Devices

(b) Emulated virtual adaptive disk.

Figure 2: Integration of an adaptive data organization mechanism realized within the hypervisor.

Paravirtualized Adaptive File System. In the case that the described adaptive data organization mechanism is implemented as an adaptive file system, the information about the current mapping of guest data to blocks is exploited by passing it through to the guests as a paravirtualized file system. This permits the guest applications to access it like a local file system. Besides this, access hints and requirements can be expressed on the basis of files and directories. If the adaptive file system is implemented within the HV as depicted in Fig. 2(a), the fact that guests and the HV are run on the same hardware can be exploited [JVHLP10] for a more efficient implementation by omitting networking facilities required to access a remote file system. Then, calls of the file system primitives as well as access hints and requirements are transported between a client component on the guest side ('FS Client') over a paravirtual I/O bus (e.g., *VirtIO* [Rus08]) to a server component ('FS Server') on the HV side using a (lightweight) file access protocol (e.g., an extension of *Styx* [PR99]). After that, the FS server applies the received commands to the adaptive file system. Alternatively, the adaptive file system can be implemented on a separate storage server and provided to the guests by the HV over a network. In this case, the FS server is located on the storage server and a FS client communicates with it through a virtual network interface employing an appropriate protocol stack. However, an issue is that current virtualization solutions do not provide the possibility to boot directly from a paravirtualized file system. This can be solved by extending the VM's firmware or by booting the guest OS from a virtual disk. Moreover, as for all paravirtualized functions, an appropriate driver is required for the guest OS.

Virtual Adaptive Disk. As mentioned in Sect. 4.1, the proposed adaptive mechanism can also be realized on the block level. However, this leads potentially to worse results because information about the page cache and about the mapping of application data to blocks is not available. Nevertheless, the mechanism can be implemented in a block-based manner on the HV and integrated as a virtual adaptive disk which is shown in Fig. 2(b). In this case, the guest application data is organized by a standard (unmodified) file system on the guest side that is established on top of a virtual adaptive disk. Such a disk can either be emulated or paravirtualized and is backed by the storage capacity of physical storage devices available to the HV. If it is emulated, a block-oriented standard disk interface (using ATA or SCSI command set) can be employed. However, it would be advantageous to provide block access hints and performance/reliability requirements to the virtual adaptive disk in order to facilitate adaptations. An opportunity to provide at least access hints through standard disk interfaces is in preparation by the responsible technical committees (*LBA access hints*). Passing access hints and performance/reliability requirements to a paravirtualized virtual adaptive disk can be realized easier due to an already (for paravirtualization) modified interface. Besides an realization within the HV, the adaptive disk can be implemented on a remote storage server. Then, a VM can either directly import the adaptive disk from the storage server over a network (e.g., SAN) or via the HV. In the latter case, the HV is responsible for the transmission of the corresponding commands, access hints, and the data blocks between a (emulated or paravirtualized) virtual disk provided to the guest and the adaptive disk on the storage server.

6 Conclusions and Future Work

Secondary storage systems for virtualized environments experience varying workloads due to a changing set of VMs and, therefore, applications that run inside them. Thus, it is appropriate to cope with this by employing adaptive storage systems. Unfortunately, existing approaches do not fully exploit all information about the characteristics of storage devices, the state of the memory system, and application requirements. Moreover, the performed adaptations are limited due to coarse-grained policies. In this paper, we tackled these limitations by introducing a mechanism that permits an adaptive reorganization of data in small steps. The proposed approach is based on an extended view of the system. We conducted a case study showing the advantages and benefits of fine-grained data layout adaptations. Additionally, we described alternative options to integrate our approach into existing HVs. However, the integration requires to extend the storage interfaces to be able to provide the proposed adaptive storage system with the necessary information about application requirements and access hints. We plan to extend our work in several directions: First, we intend to develop a cost model that permits to perform reorganization steps based on a close to reality cost/benefit comparison. Based on this, we want to investigate reorganization strategies in order to find out which type of reorganization represents the most effective way to optimize the data organization in the face of a particular change of application needs and workload. Furthermore, we plan to implement the proposed adaptive storage system and evaluate it under synthetic benchmarks as well as under workloads captured from example applications commonly used in virtualized environments. Finally, we also intend to integrate a prototypical implementation of our approach into a HV such as KVM/QEMU.

References

[AvMT10] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Block-level RAID is dead. In *Proceedings of the 2nd USENIX conference on Hot topics in stor-*

age and file systems, HotStorage'10, pages 4–4, Berkeley, CA, USA, 2010. USENIX Association.

[AvMT12] Raja Appuswamy, David C. van Moolenbroek, and Andrew S. Tanenbaum. Integrating flash-based SSDs into the storage stack. In *MSST*, pages 1–12. IEEE, 2012.

[BN12] Anirudh Badam and David W. Nellans. Enabling Application Directed Storage Systems. In *Proceedings of the 3rd Annual Non-Volatile Memories Workshop*, NVMW 2012, San Diego, CA, USA, March 2012.

[CKZ11] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Hystor: making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 22–32, New York, NY, USA, 2011. ACM.

[GPG$^+$11] Jorge Guerra, Himabindu Pucha, Joseph Glider, Wendy Belluomini, and Raju Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX conference on File and stroage technologies*, FAST'11, pages 20–20, Berkeley, CA, USA, 2011. USENIX Association.

[JMBR11] Nikolaus Jeremic, Gero Mühl, Anselm Busse, and Jan Richling. The pitfalls of deploying solid-state drive RAIDs. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*, pages 14:1–14:13. ACM, June 2011.

[JVHLP10] Venkateswararao Jujjuri, Eric Van Hensbergen, Anthony Liguori, and Badari Pulavarty. VirtFS - A virtualization aware File System passthrough. In *Proc. Ottawa Linux Symposium*, 2010.

[KGP89] R.H. Katz, G.A. Gibson, and D.A. Patterson. Disk system architectures for high performance computing. *Proceedings of the IEEE*, 77(12):1842–1858, 1989.

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *SIGMOD Rec.*, 17:109–116, June 1988.

[PR99] Rob Pike and Dennis M Ritchie. The Styx® architecture for distributed systems. *Bell Labs Technical Journal*, 4(2):146–152, 1999.

[Rus08] Rusty Russell. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[SDH$^+$96] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 1–1, Berkeley, CA, USA, 1996. USENIX Association.

[SWZ98] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. Data partitioning and load balancing in parallel disk systems. *The VLDB Journal*, 7(1):48–66, February 1998.

[WGSS96] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Trans. Comput. Syst.*, 14(1):108–136, February 1996.

[WR09] Xiaojian Wu and A.L.N. Reddy. Managing storage space in a flash and disk hybrid storage system. In *Modeling, Analysis Simulation of Computer and Telecommunication Systems, 2009. MASCOTS '09. IEEE International Symposium on*, pages 1–4, 2009.

[YVZS09] Chaitanya Yalamanchili, Kiron Vijayasankar, Erez Zadok, and Gopalan Sivathanu. DHIS: discriminating hierarchical storage. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 9:1–9:12, New York, NY, USA, 2009. ACM.