Supporting and Exploiting Heterogeneity in the Storage Stack

Raja Appuswamy, David C. van Moolenbroek, Andrew S. Tanenbaum
Dept of Computer Science,
Vrije Universiteit,
{raja, dcvmoole, ast}@cs.vu.nl

Abstract

With the widespread adoption of NAND-flash-based SSDs as a primary storage medium, heterogeneity in storage installations has become a norm rather than an exception. In this paper, we argue that the compatibility-driven traditional storage stack is fundamentally flawed and incapable of supporting and exploiting heterogeneity properly. After highlighting major issues with the traditional stack, we show how Loris, our fresh redesign of the storage stack, supports easy integration of heterogeneous devices with minimal development effort. We also show how Loris, with a few extensions, can be transformed into a flexible, modular framework for implementing several hybrid configurations that exploit heterogeneity in the storage stack.

1 Introduction

Over the past few years, the storage hardware landscape has changed considerably. New devices, with interfaces and price/performance/reliability characteristics strikingly different from conventional block-based disk drives, have necessitated rethinking several aspects of storage management. NAND flash-based SSDs have bought about a sea change in the design of storage systems. Several academic and industrial research projects have proposed using SSDs in different hybrid configurations with disk drives to maximize benefits in terms of price, performance, or reliability. In short, supporting heterogeneous device types, and exploiting heterogeneity using hybrid configurations has become an extremely important topic of research.

In this paper, we argue that the traditional storage stack has several fundamental design issues that undermine its effectiveness in supporting and exploiting heterogeneous installations. In prior work, we presented Loris [2], our fresh redesign of the traditional storage stack that solves several reliability, flexibility, and heterogeneity issues by design. In this paper, we show how 1) the modular division of labor in Loris supports easy integration of heterogeneous devices, 2) our new storage model simplifies administration of these devices, and 3) we use device-awareness and file-awareness together to support several hybrid configurations that exploit heterogeneity in the Loris stack.

The rest of the paper is organized as follows. In Sec. 2, we explain why heterogeneity must be considered a first-class citizen in system design and we detail the shortcomings of the traditional storage stack. In Sec. 3, we present a quick overview of Loris. We then detail the infrastructure support for exploiting heterogeneity in Sec. 4, following which, we present a sample hybrid storage configuration in Sec. 5 and conclude in Sec .6.

2 Heterogeneity as a first class citizen in storage system design

In this section we will show how device and application diversity creates several heterogeneity issues for the traditional storage stack.

2.1 Device diversity: differences across device families

New devices, with interfaces different from the traditional block-based read/write interface, are emerging in the storage market. Some flash devices, for instance, are byte accessible, and have an erase operation in addition to read/write operations. Two approaches have been taken toward integrating these devices into the traditional storage stack.

The first approach involves writing new file systems for each new device type. The file system communicates directly with the device and provides device-specific layout algorithms. This approach however renders the file system incompatible with the traditional block-based RAID and volume management implementations. As a result, one needs to reimplement these functionalities again for each device type, resulting in a different storage stack per device family.

The second approach involves adding an additional level of indirection, as seen with flash translation layers. This layer abstracts device-specific interfaces and access restrictions by exposing the traditional block interface and transparently mapping blocks to device-specific abstractions. Even though this approach retains backward compatibility with existing installations, it widens what has been referred to as the *information gap*—multiple layers in the stack (the file system, and the translation layer) performing independent optimizations unaware of the effect they have on other layers. For instance, all layout optimizations performed by the file system are rendered futile, or might even degrade performance as the translation layer employs its own algorithm for mapping file system blocks to device-specific abstractions. Thus, the traditional stack fails to support heterogeneous storage devices without a significant redesign.

2.2 Device diversity: Heterogeneity within device families

It is well known that SSD firmware design offers several parameters that can be tweaked to achieve different price/performance/reliability trade-offs [1]. As a result, different SSDs, sometimes even from the same vendor, have different performance characteristics. For instance, Intel X25-V SSD design makes a price/performance trade-off, as it sacrifices sequential throughput by reducing the number of channels populated with NAND flash. Intel X25-M on the other hand, has equally impressive random and sequential read/write performance figures but costs more than its economical counterpart. The traditional storage stack fails to exploit this heterogeneity to achieve better performance or reliability.

Exploiting heterogeneity in storage installations requires pairing devices with their ideal layout algorithms. For instance, a log-structured layout might be best suited for an Intel X25-M, while it could deteriorate performance when used with X25-V due to the unnecessary cleaning overhead. As file systems handle device-specific layout in the traditional stack, this pairing translates to a "one file system per device" bond. However, logical volume managers intentionally broke this very bond to simplify storage management and administration by creating a "one file system per logical device" bond. As a result, multiple file systems with independent layout algorithms could share the same physical device thereby rendering all layout optimizations pointless. Thus, the traditional stack fails to support device heterogeneity both within and across device families.

2.3 Application diversity

Like devices, applications also vary widely in their access patterns. While some files, like log fies, are append-only, others might be read and written in their entirety. Some files might be accessed sequentially, while others, like database tables, might be accessed randomly. Users also tend to associate policies and importance levels with files or file types. For instance, while a user might want to protect his photographs against multiple device failures, he might be less concerned about compiler temporaries. An ideal storage stack should exploit this diversity by matching file properties with device properties, and provide redundancy on a per-file basis.

Due to the compatibility-driven integration of storage virtualization at the block level, volume managers are unaware of files. Due to the integration of RAID algorithms at the block level, redundancy policies can be applied only on at the coarse granularity of volumes. Thus, the semantically-unaware, block-based volume management layer cannot be used to exploit heterogeneity. As traditional file systems work based on the "one file system per logical device" bond, they are incapable of managing multiple devices. Furthermore, file systems map entire subtrees to a single physical or logical device, and as a result, all files in a file system use the same policies and layout algorithms. Thus, the file systems are also incapable of exploiting device heterogeneity and application diversity.

3 Background: The Loris storage stack

In prior work, we highlighted several issues that plague the traditional storage stack. We proposed Loris [3, 2], a fresh redesign of the stack and showed how the right division of labor among layers in Loris solves all problems by design. In this section we provide a brief overview of Loris.

Loris is made up of four layers as shown in Figure 1. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Every Loris file is uniquely identified using a <volume identifier, file identifier> pair. Each Loris file is also associated with several *attributes*, and the interface supports attribute two manipulation operations—*getattribute* and *setattribute*. Attributes enable information sharing between layers, and are also used to store out-of-band file metadata. We will now detail the abstraction boundaries and responsibilities of each layer in a bottom-up fashion.

3.1 Physical layer

The physical layer is tasked with three primary responsibilities. The physical layer exports a "physical file" abstraction to the logical layer. The logical layer stores data from both end-user application, and other Loris layers in physical files. Thus, the first responsibility of the physical layer is providing device-specific layout schemes, and persistent storage of files and their attributes. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*. Each device, and hence its physical module, is uniquely identified using a *physical module identifier*. Our prototype physical layer was based on the original MINIX 3 file system layout scheme.

The second responsibility of this layer is providing data verification. Each physical layer implementation must support some comprehensive corruption detection technique, and use it to verify both data and metadata. As the physical layer is the lowest layer in the stack, it verifies requests from both applications and other Loris layers alike, thereby acting as a single point of data verification. Our prototype physical layer supports parental checksumming [4] and uses it to provide end-to-end data integrity.

The third responsibility of the physical layer is supporting fine-grained data sharing and individual file snapshoting. By delegating support for file snapshoting to the physical layer, the Loris stack maximizes storage efficiency without performance degradation. We have implemented physical layers that support both copy-based, and copy-on-write based snapshoting in our Loris prototype [3].

3.2 Logical layer

The logical layer provides both device and file management functionalities. The logical layer can be considered to be made up of two sublayers, namely the file pool sublayer

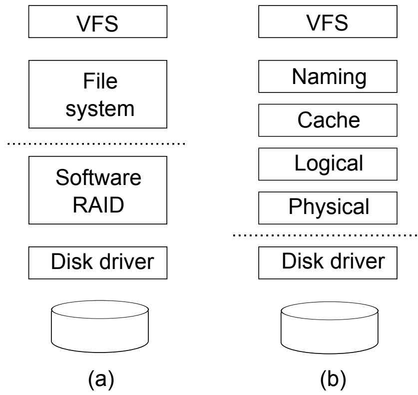

Figure 1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

at the bottom, and the volume management sublayer at the top. The logical layer exports a *logical file* abstraction to the cache layer. A logical file is a virtualized file that appears to be a single, flat file to the cache layer. Details such as the physical files that constitute a logical file, the RAID levels used, etc. are confined within the two sublayers. We will now briefly describe the functionalities of each sublayer.

3.2.1 File pool sublayer

In prior work, we presented *File pools* [3], a new storage model based on the Loris stack. File pools are the unit of storage administration. An administrator creates a file pool by specifying the set of devices that constitute the pool. New devices can be added to, and old devices be removed from, existing file pools. When a new device is added to a file pool, a physical module suitable for that device is automatically started. This new physical module registers with the logical layer, and after registration, the logical layer can start forwarding new file create requests, or migrate existing files, to the new physical module. Similarly, when a device needs to be removed from a file pool, the logical layer simply migrates the files from that device to a new spare, or distributes them among existing devices. Thus, file pools automate several administrative operations.

The file pool sublayer provides support for realizing the file pool storage model. It maintains data structures necessary for tracking device memberships in file pools, and provides device management operations (like adding/removing devices). It also exposes a file create/delete interface to the volume management sublayer, and is responsible for satisfying file allocation requests. We implemented a primitive load balancing algorithm in our prototype that just rotates file allocation requests across the physical modules that constitute a file pool.

3.2.2 Volume management sublayer

The volume management sublayer is responsible for providing both file volume virtualization and per-file RAID services. Multiple file volumes can be created in a single file pool in Loris. Thin provisioning of file volumes is possible as all file volumes share the same pool of files, and no space needs to be reserved up-front during volume creation time.

Each file volume is represented by a *volume index file* that tracks logical files belonging to that volume. The volume index is created during volume creation, and it stores an array of entries containing the *configuration information* for each logical file in that volume. This configuration information is 1) the RAID level used, 2) the stripe size used, and 3) the set of physical files that make up the logical file, each specified as a <physical module identifier, inode number> pair. For instance, a logical file belonging to a volume V1, with identifier F1, that is backed by an inode I1 on physical module D1, would have F1=<raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I1>> as its configuration information. Similar to the way files are tracked by the volume index file, file volumes themselves are tracked using the *meta index file*. This file also contains an array of entries, one per file volume, containing *file volume metadata*. Thus, using these two data structures, the volume management sublayer supports file volume virtualization.

Per-file RAID services provided by this sublayer utilize the file allocation services of the file pool sublayer to group multiple physical files in a redundant configuration. For instance, when the logical layer receives a file allocation request, it is forwarded to the volume management sublayer. This sublayer then invokes the file create operation provided by the file pool sublayer, passing the required number of physical files as a parameter. The file pool sublayer returns back the <physical module identifier, inode number> pairs after creating physical files on the corresponding physical modules, and the volume management sublayer records these pairs in the volume index file of the corresponding file volume.

3.3 Cache and Naming layers

The cache layer provides data caching. As the cache layer is file-aware, it can provide different data staging and eviction policies for different files. Fully exploiting heterogeneity requires multiple layers in the stack to act in concert, and file-awareness in the cache layer makes it possible to pick policies based on file access patterns, as we will see in the next section.

The naming layer acts as the interface layer. Our prototype naming layer implements the traditional POSIX interface. The naming layer uses Loris files to store data blocks of directories that contain directory entries. It also uses the attribute infrastructure in Loris to store POSIX attributes of each file as Loris attributes. All POSIX semantics are confined to the naming layer. For instance, as far as the logical layer is concerned, directories are just regular files that should be replicated on all local devices. By confining interface semantics to the naming layer, we make it possible to replace it, with a search-based one for instance, without affecting any other layer.

4 Supporting hybrid configurations with Loris

In this section, we will describe infrastructure support for exploiting heterogeneity in the storage stack. We will first show how Loris supports easy integration of heterogeneous devices with its modular division of labor. We will then introduce our plugin based infrastructure for exploiting heterogeneity, and describe in detail the roles and responsibilities of each plugin. Sec. 5 presents sample hybrid configurations and the algorithms we implemented for each plugin in detail.

4.1 Pairing devices and layout algorithms

Supporting heterogeneity in the storage stack requires the ability to pair layout algorithms with devices without affecting RAID and volume management functionalities. As the physical layer in Loris exports a physical-file abstraction, all device-specific interfaces and access restrictions are confined to the physical layer. Since the logical layer works with physical files, volume management and RAID functionalities provided by the logical layer are device independent. Thus, integrating new device types into the storage stack requires designing and implementing just a new physical module. The naming, cache, and logical layer implementations can be shared across device families.

Since each device gets its own physical module during initialization time, it is possible to use different physical modules, hence different layout algorithms, without affecting any of the higher layers. Thus, one could use a log-structured layout on an disk drive and a read-optimized layout on an X25-V, thus pairing devices with their ideal layout algorithms. Thus, Loris supports heterogeneity both within and across device families.

4.2 Device types and device classification plugin

Exploiting heterogeneity in a storage installation requires the stack to be aware of device properties. The *device classification plugin* monitors devices for certain properties and uses them to assign a *device type* to each device. A device type is a collection of *device properties*. Each device property specifies an aspect of a device that can be used to make a policy decision. Some examples property classes include performance, reliability, and power consumption.

Device properties can be gathered using two approaches. In the static approach, these properties are gathered once, when a device is added to a file pool. In the dynamic approach, devices can be probed regularly to collect dynamic power/reliability metrics, and file operations issued to each physical module can be monitored to collect dynamic performance metrics. For instance, read/write response times observed at run time for each file read/write operation can be used to derive an average response time using exponential averaging [5]. A subtle but important point to note is that, in both the static and dynamic cases, unlike reliability/power consumption metrics, performance statistics are not gathered on the raw device, but on the device-physical module combination. Since each layout algorithm has the potential to alter device performance, we believe that measuring the effective performance of the pair provides a more accurate estimate of expected performance at the logical layer.

4.3 File types and type assignment plugin

We modified Loris to monitor file access patterns at run time. We decided to monitor file accesses at the cache layer, for two reasons: 1) to respond quickly to changes in access patterns, and 2) to pick different caching algorithms for files with different access patterns, as using the same staging and eviction policies for all files might pollute the cache and reduce its effectiveness in satisfying cached reads. For instance, consider a file that is always randomly read in small chunks. A dumb cache would always prefetch data blocks for the file during each read request, resulting in potentially useful blocks being evicted from the cache to accommodate these useless new blocks.

We collects statistics at the cache layer about each file during each *Loris session*. A Loris session for a file F is the duration between the first request for F, and the last eviction of Fs data pages from the cache. Within a Loris session, we collect statistics like read/write counts, read/write sequentiality etc. At the end of each session, the plugin uses the statistics collected to assign a file type to each file.

A file type is nothing but a collection of *file properties*. Each file property specifies an aspect of a file that can be used to make a policy decision. While some properties are numeric, others just classify a file as belonging to a particular property class. For instance, while aggregate read/write ratio is a numeric property, read sequentiality property just species whether the file is read sequentially or randomly in large or small chunks. We refer to this process of using session statistics to assign types as *measurement-driven type assignment*. We would like to point out that other techniques that perform post-processing of trace logs for making long-term migration decisions, or periodic reassessment of current allocation, are complementary to measurement-driven type assignment.

Since the type assignment plugin is completely responsible for interpreting the semantics of each type, it is possible to extend file types, or add new file types based on other characteristics like user-requested importance level. The file type determined by the plugin is shared with the logical layer using our attribute infrastructure. The volume index file maintained by the volume management sublayer stores a *type table* for each file, in the form of a circular list that maintains the last N type assignments. We refer to the most common file type in the type table as the *active file type*. The reason behind storing the last N type assignments is to weed out short-term anomalies in access patterns.

4.4 Type mapping plugin

If adding the new file type to the type table results in a change in the active file type, the logical layer invokes the *type mapping plugin* to initiate a *type casting* operation. As the name indicates, type casting refers to the process of changing a file

Perf metric	Device1	Device2	Device3
SEQUENTIAL READ LIST	READ-OPTIMIZED HDD	WRITE-OPTIMIZED HDD	SSD
SEQUENTIAL WRITE LIST	WRITE-OPTIMIZED HDD	READ-OPTIMIZED HDD	SSD
RANDOM READ LIST	SSD	READ-OPTIMIZED HDD	WRITE-OPTIMIZED HDD
RANDOM WRITE LIST	WRITE-OPTIMIZED HDD	SSD	READ-OPTIMIZED HDD

Table 1: Performance lists maintained by module assignment plugin for the hypothetical storage configration specified in Sec. 4. The first column specifies the performance metric for which the list is being maintained. The rest of the columns list devices in decreasing performance order.

from one type to another. A type casting operation involves three steps:1) determining the ideal device type for the new file type, 2) calling the module assignment plugin to determine new physical modules, and finally 3) queuing the file for migration with the migration plugin. As the name indicates, the type mapping plugin is responsible for the first step. It is the only plugin that needs to be aware of both file and device types as it acts as a link between other plugins. The type mapping plugin can use several file access statistics like recency of access, or frequency of access, and other device characteristics, like power consumption to deploy hybrid configurations with widely varying trade-offs.

4.5 Module assignment plugin

Once the type mapping plugin determines the device type, the *module assignment plugin* is invoked to assign a new set of physical modules corresponding to the new device type. This plugin is just device type-aware. It could use any device property or a combination of several properties to assign physical modules. For instance, a physical module could pick candidates based on aggregate read/write response times, and use device power consumption as a secondary factor in determining the target physical module. We are exploring several dynamic performance metrics like read/write response times, space utilization summaries, device heat metrics etc as a part of future work. To support new metrics, one needs to update only the device classification and module assignment plugins to gather metrics, and include them in decision making.

4.6 File migration plugin

After the module assignment plugin picks the new target, the file is queued for migration with the *file migration plugin*. The migration plugin works transparently in the background and performs incremental migration of file data. It first takes a snapshot of the target file using the individual file snapshotting functionality present in Loris. Following this, the migration plugin creates new physical files on the destination physical modules, and copies over data and attributes from the source modules to the new destination. After successfully completing the transfer, the migration plugin checks to see if the file has been modified after the snapshot. If so, it performs an incremental transfer of just the modified data.

As a part of supporting efficient deletion of file volume snapshots, we added an explicit call between the logical and physical layers that makes it possible for a physical layer implementation to share a physical file's unshared block offsets with the logical layer. Unshared blocks are those blocks pointed to by an inode that are not shared with any other snapshot inodes. The logical layer uses this call to determine the blocks of a snapshot volume's volume index file that were modified in that snapshot. Armed with this knowledge, the logical layer just processes the logical file entries contained in those blocks, speeding up snapshot deletion significantly as it can avoid checking all other files in the snapshot volume. The migration plugin uses the same call to determine the incremental set of blocks to migrate. This migration process continues until there is no data left to migrate. If migration is successful, the old physical files are deleted, and the file's logical configuration is updated in the volume index to point to the new physical files.

5 Hybrid configurations

In this section, we will present a sample hybrid configuration. For this configuration, we consider a hypothetical installation with an SSD and two HDDs. We assume that 1) our hypothetical SSD provides better IOPS than HDDs for random reads/writes, but the HDDs provides better sequential read/write throughput, and 2) as a result of better sequential write performance, a HDD using a write-optimized layout performs better than an SSD for random/sequential write configurations.

We believe both these assumptions are valid for two reasons. First of all, it is very common to find enterprise and consumer-grade HDDs that have better sequential read/write performance than some of their SSD counterparts. Several related projects have also made this assumption in designing allocation and migration algorithms for heterogeneous installations containing HDDs and SSDs [5]. Secondly, poor random write performance in SSDs is a well studied research problem [1]. The range of options available during SSD firmware design can lead to widely different design benefits. Intel X25-V, for instance, provides very good random-write performance (40 MB/s random write throughput). But a write-optimized layout on a HDD could easily provide as much as twice the performance of X25-V for random write workloads in ideal conditions (90 MB/s or more with a modern SATA disk). To exploit this heterogeneity, we pair a read-optimized physical module with the SSD and one HDD, while the other HDD is paired with a write-optimized physical module.

File type	Device type mapping	Device assigned
size = SMALL + rwpattern = MOSTLY_R	readtype = RAND	SSD
size = SMALL + rwpattern = MOSTLY_W	writetype = RAND	WRITE-OPTIMIZED HDD
size = SMALL + rwpattern = RW	readtype = RAND + writetype = RAND	SSD
size = LARGE + rwpattern = MOSTLY_R + accesstype = SEQ	readtype = SEQ	READ-OPTIMIZED HDD
size = LARGE + rwpattern = MOSTLY_R + accesstype = RAND	readtype = RAND	SSD
size = LARGE + rwpattern = MOSTLY_W + accesstype = SEQ	writetype = SEQ	WRITE-OPTIMIZED HDD
size = LARGE + rwpattern = MOSTLY_W + accesstype = RAND	writetype = RAND	WRITE-OPTIMIZED HDD
size = LARGE + rwpattern = RW + accesstype = SEQ	readtype = SEQ + writetype = SEQ	WRITE-OPTIMIZED HDD
size = LARGE + rwpattern = RW + accesstype = RAND	readtype = RAND + writetype = RAND	SSD

Table 2: Hybrid configuration using an SSD and two HDDs. The first column lists all file types belonging to the type taxonomy used by our type assignment plugin. The second column lists device types generated by the type mapping plugin corresponding to each file type. The third column lists the target device chosen by the module assignment plugin.

5.1 Base configuration

We will now describe in detail the algorithms and data structures used by each plugin for implementing our first hybrid configuration.

5.1.1 Device classification plugin

Our device classification plugin gathers four performance metrics for each device, namely, the sequential read/write throughput, and random read/write IOPS, when the device is added to a file pool. It maintains four in-memory *performance lists*, one corresponding to each performance metric. These lists link data structures representing physical modules corresponding to each device in descending performance order, as shown in Table 1.

5.1.2 Type assignment plugin

The first column in Table 2 shows the file type taxonomy that is used by our type assignment plugin. As can be seen, each file type consists of three file properties, namely, size, read/write pattern, and access type. The size property classifies the file as small, medium or large. The read/write pattern specifies whether the file is read mostly, or written mostly or both read and written. The access type property specifies whether the file is read/written sequentially or randomly. The type assignment plugin uses the access statistics collected in each session to classify a file into one of the types listed in the table.

5.1.3 Type mapping plugin

The second column in Table 2 shows the device type generated by our type mapping plugin for each file type. As can be seen, small files are mapped to devices that excel in random access performance while large files use a straightforward mapping based on accesstype. Partitioning small files into read-mostly and write-mostly results both file types being assigned to ideal physical modules for maximizing performance gains.

5.1.4 Module assignment plugin

The third column in Table 2 shows the physical module-device pair picked by our module assignment plugin. This plugin uses the device type generated by the type mapping plugin to determine the performance lists to be searched for picking a new physical module. For instance, consider a request for a device type with the following properties: <readtype = RAND, writetype = SEQ>. The module assignment plugin first walks through the random read IOPS list, assigning each physical module a numerical score equal to its position in the list. Following this, it walks through the sequential write throughput list, updating the module scores with their offset in this second list. Finally, it picks the module with the lowest score as the target module. If more than one module gets the same score, the algorithm gives modules with better read performance a priority.

6 Conclusion

In this paper, we outlined several fundamental issues that undermine the effectiveness of the traditional storage stack in supporting and exploiting heterogeneity. We then presented Loris, our fresh redesign of the storage stack. We showed how Loris, with minor extensions, can be used as a framework for supporting and exploiting heterogeneity.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *USENIX Ann. Tech. Conf.*, pages 57–70. USENIX Association, 2008.

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Loris - a dependable, modular file-based storage stack. In *Proc. of the The 16th IEEE Pacific Rim Intl. Symp. on Dependable Computing*, pages 4–4. USENIX Association, 2010.

[3] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Flexible, Modular File Volume Virtualization in Loris. Technical Report IR-CS-67, January 2011.

[4] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Parity lost and parity regained. In *Proc. of the Sixth USENIX Conf. on File and Storage Technologies*, pages 1–15. USENIX Association, 2008.

[5] X. Wu and A. 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.