DHIS: Discriminating Hierarchical Storage

Appears in the Proceedings of the Israeli Experimental Systems Conference (ACM SYSTOR '09)

Chaitanya Yalamanchili, Kiron Vijayasankar, and Erez Zadok
Stony Brook University

Gopalan Sivathanu
Google Inc.

ABSTRACT

A typical storage hierarchy comprises of components with varying performance and cost characteristics, providing multiple options for data placement. We propose and evaluate a hierarchical storage system, DHIS, that uses application-level hints to discriminate between data with different access characteristics, and then customizes its placement and caching policies to each type. The data placement decisions in DHIS are made in an online fashion, during data creation. Most existing solutions that attempt to customize data layout require moving data around, based on access characteristics. DHIS uses two kinds of information to make its decisions. First, it uses knowledge about higher-level *pointers* between blocks (for example, file system pointers) to understand the relationship between blocks and consequently, their importance. Second, DHIS defines a set of generic attributes that the higher layers can use to annotate data, conveying various properties such as importance, access pattern, etc. Based on these attributes, DHIS dynamically decides to place the data in the hierarchy best suited for its requirements. By doing so, DHIS solves a critical problem faced by storage vendors and developers of higher level storage software, in terms of choosing the most efficient policy among many alternatives. Through several benchmarks, we show that DHIS's data placement decisions improve performance significantly.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management—*Storage hierarchies*; D.4.2 [**Operating Systems**]: Storage Management—*Allocation/deallocation strategies*; C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Design, Reliability, Measurement, Performance

Keywords

Storage Stack, Intelligent Disks, File Systems, Storage Systems

1. INTRODUCTION

Modern large storage systems are virtually supercomputers; a typical high-end storage system from EMC [7] has hundreds of processors, tens of gigabytes of RAM and hundreds of disks. In tune with the increasing processing power available at the storage systems, their functional sophistication has also increased. Today, storage systems employ various forms of RAID for reliability and performance, use non-volatile RAM to absorb write latency, perform dynamic block migration for load balancing, etc. [7, 20].

Although storage systems have evolved significantly in terms of the range of functionality they provide, they are still constrained due to one fundamental limitation: they have little or no information about the system layers above that use the storage system, and thus view data simply as a flat stream of bytes. Although some large-scale storage systems such as the NetApp FS6000 [15] understand higher-level file structures, and export a file-system–like interface, a significant number of others still support a generic block-based interface for reasons of flexibility [7]. Such storage systems for example do not know the relative importance of data, their access-patterns, etc. Although a lot of storage-level policies such as RAID level, caching policy, etc. can be tuned for specific kinds of usage, a typical storage system cannot fully exploit this potential because it deals with a myriad of interleaved types of data each with different access characteristics, and has very little information to separate these types from each other.

In this paper, we present a **D**iscriminating **Hi**erarchical **S**torage system, DHIS (pronounced as *this*), that uses various hints specified from the higher layers about the type of the data to select custom policies for managing the data, such as the exact RAID level, cacheability of the data in NVRAM, etc. DHIS also uses information on the logical relationship between blocks conveyed in the form of logical pointers [16] to extrapolate its type information from one identifying block to its descendants. By being able to discriminate between data with varying requirements, DHIS is able to balance conflicting goals such as performance and reliability much more efficiently than traditional storage systems.

To make informed choices on the exact layout and caching policies to use for a specific piece of data, DHIS enables the layers above to annotate logical chunks of data with *attributes* on the data. For instance, the file system can specify that a given file (identified by the top-level inode block for the file) will be mostly subject to small random writes. If this attribute is associated with the file, DHIS can ensure to not place the file in a RAID-5 format, given the small-write performance penalty incurred in RAID-5; instead, it may choose to place it in RAID-1 (mirroring) format.

DHIS supports five attributes:

- Importance of the data (which determines how reliably the data should be stored).

- The normal access-pattern on the data (i.e., random or sequential).
- The expected popularity of the data (i.e., hot or cold).
- Whether the data is read-mostly or write-mostly.
- The expected lifetime of the data (i.e., whether it corresponds to a temporary file).

Based on these five attributes, DHIS decides on the specific redundancy and reliability scheme to use for the data, and the various forms of caching to use (e.g., whether to cache the data in NVRAM or perhaps a faster Flash storage layer) such that the best performance/reliability trade-off is obtained. Specifically, the current implementation of DHIS utilizes these attributes to automatically select the RAID level a piece of data goes into, and to decide which pieces of data to cache in NVRAM.

We evaluate DHIS using a software prototype implementation in the Linux kernel. Our prototype operates as a pseudo device driver that interposes between the file system and the software RAID layers. One key challenge in this prototyping environment is to ensure there is no performance interference between the host application and the processing at the pseudo driver layer. By careful use of kernel isolation techniques, we separate the CPU's and memory's usage of the software prototype from the host applications, thus providing a fairly close approximation of an actual hardware prototype with its own processing and memory. We believe that this prototyping environment is valuable more generally for evaluating other kinds of functionality in the storage system.

Using this prototyping environment we evaluate the various discriminating policies of DHIS and demonstrate their effectiveness. We show that DHIS can achieve significant performance wins by exploiting higher-level attributes. We show that the flexibility to choose RAID levels on a per-file basis provides significant benefits in performance, compared to the one-size-fits-all solution normally employed in today's systems. We also show that by intelligent NVRAM caching of data that is subject to frequent random writes (e.g., meta-data blocks in a file system), DHIS greatly improves overall system performance. As we see in Section 5.4.4, caching meta-data selectively in NVRAM can improve write performance by 37% for random I/O-intensive workloads.

Overall, we find that DHIS offers an interesting design choice for building storage systems that exploit higher level system information. By allowing the higher layers such as the operating system to express attributes inherent only to the data and not what the storage system should do with it, we decouple the layers; in other words, the file system need not understand the specifics of the wide variety of low-level mechanisms and policies that today's storage systems use. Depending on the specific features available within a storage system, the storage system can decide how to exploit this valuable extra information.

The rest of this paper is organized as follows: in Section 2, we discuss the background of modern storage systems and type-aware storage. In Section 3, we describe the design details of DHIS and show the kind of optimizations that DHIS enables. Section 4 presents our disk prototyping framework and our prototype implementation of DHIS. We evaluate our prototyping framework and our implementation in Section 5. We discuss related work in Section 6 and conclude in Section 7. Finally, in Section 8, we discuss how we plan to extend this work in the future.

2. BACKGROUND

In this section, we first describe the current state of the art in hierarchical storage and motivate the need for fine-grained policies specific to data. We then briefly discuss the usage of pointer information in type-safe storage which our work builds upon.

2.1 Modern Large-Scale Storage Systems

Large-scale storage systems today comprise diverse resources that include high processing power, hundreds of gigabytes of RAM, solid state storage media such as flash, and hundreds or even thousands of disks [7]. Modern storage systems run complex software to provide functionality such as reliability, fault-tolerance, and high performance I/O. One of the challenges in such storage systems is to effectively manage the wide range of resources to provide optimal performance and customizable features. However, despite the advancement in storage hardware, the interface used for communicating with them is still simple and narrow in most scenarios. For example, the SCSI interface supports just two main primitives, block `read` and `write`, resulting in the storage system being mostly oblivious to higher-level information. This makes efficient resource management within modern storage systems a difficult problem, as storage systems cannot discriminate between different kinds of information they store.

Some existing systems try to work around this problem by exporting more information to higher-level software [6, 9]. For example, certain enterprise-class storage systems allow higher-level software to choose the RAID level to use for a new volume, during creation [10]. However, this requires that the file system or higher-level storage software be aware of the characteristics of each volume, which could be totally tied to the internal architecture of the specific storage systems. For example, a storage system could contain several fine-grained RAID levels, as well as devices such as NVRAM and solid state memory. Storage architectures could also be different across vendors and models, and it may be cumbersome to customize file systems for specific storage systems. Moreover, the abstraction of a volume is in most cases too coarse-grained to express difference in access characteristics across files. Therefore, it is more flexible and straight-forward to communicate the attributes associated with data to the storage system, and let the storage system decide the policies based on its internal architecture.

2.2 Type-Awareness in Storage

Recently, there have been efforts to bridge the information gap between storage systems and higher level layers using Object-Based Storage devices (OSD) [14] and Type-Safe Disks (TSD) [16]. While OSDs fundamentally change the view of storage systems by exporting an object interface, TSDs extend the traditional block interface by introducing the notion of logical pointers between blocks. In this section, we briefly describe TSDs which form the basis for DHIS.

TSDs aim at communicating file-system–level block pointers to the disk system, enabling useful functionality at the disk-firmware level. File system pointers indicate logical relationships between disk blocks. For example, a per-file meta-data block (such as an Ext2 inode block) has pointers to data blocks belonging to a file. At the disk-level, pointers help infer at least three key pieces of information:

1. *Logical grouping of blocks.* For example, the set of blocks pointed to by a per-file meta-data block can represent all data belonging to a file.
2. *Relative importance of blocks.* Blocks with outgoing pointers impact the reachability of one or more other blocks, and

hence are more important. For example, meta-data blocks are more important than regular data blocks.

3. *Block liveness*. Current disk systems cannot differentiate used and unused blocks. With pointers, a disk system can infer that blocks without any incoming pointers are unused (as they cannot be reached).

Information about higher-level pointers enables a disk system to provide useful functionality that cannot be provided by present day's disks. For example, a TSD can perform intelligent prefetching of blocks based on logical relationships indicated by pointers (when a meta-data block is accessed, blocks pointed to by it can be prefetched). Knowledge of relative importance of blocks can be used to adopt better redundancy schemes for important blocks.

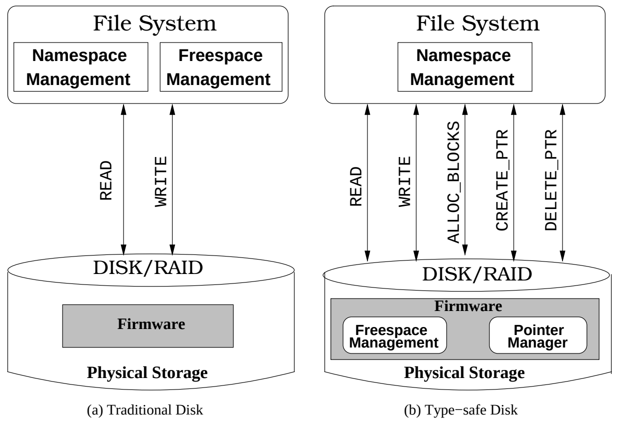


Figure 1: ***Traditional disks vs. TSDs***

.

The architectural differences between a regular disk and a TSD are shown in Figure 1. TSDs export an extended block-based interface to support pointers, and to perform free-space management at the disk level. Disk primitives to allocate blocks and adding or removing pointers can be used by higher-level software such as file systems to communicate the necessary information to the disk. TSDs also perform automatic garbage collection of blocks that are not reachable through any pointer, thereby obviating the need for the higher-level software to manage free blocks explicitly. For example, the Ext2TSD file system [16] is a modified Ext2 file system that exports pointer information to a TSD.

3. DESIGN

In this section, we describe the design of DHIS in detail and discuss the optimizations that DHIS achieves by using higher-level attributes on data. In Section 3.1, we describe the type-aware hierarchical storage setup that DHIS incorporates and its salient features. In Section 3.2, we present the set of well-defined higher-level attributes that DHIS supports. Finally, in Section 3.3, we show the kind of optimizations that these attributes enable.

3.1 A Hierarchical Storage Architecture

DHIS's architecture comprises volatile and NVRAM, as well as several individual disks aggregated using standard RAID levels. In our design, we particularly consider the three most commonly used RAID levels: RAID0 (striping without redundancy), RAID1 (mirroring), and RAID5 (striping with a parity block per stripe). These three RAID levels have varying characteristics in terms of performance, reliability, and cost per gigabyte. We aim to use these resources within a single storage system and manage them efficiently in a transparent manner using higher-level hints about data and access semantics. DHIS exports a flat namespace to higher-level storage software such as file systems, and aggregates the storage capacity available in the different RAID levels internally. The architecture we use while designing DHIS is shown in Figure 2.

In the rest of this section, we detail the basic design aspects of operating such a hierarchical storage system in a type-aware storage setup as described in Section 2.2.

3.1.1 Virtualizing the Block Layer Namespace

Although DHIS manages several disks and RAID levels internally, it appears like a single disk system to higher-level software. For this purpose, it maintains a block-address virtualization layer that contains an address translation table, `TTABLE`, which maps the global logical block namespace to individual disk-specific addresses. A physical address contains two parts: a disk or device identifier (e.g., an internal RAID device), and a physical block number within that device. The `TTABLE` is looked up for every I/O request, and is updated whenever blocks need to be re-mapped to different devices. DHIS stores the `TTABLE` and other bookkeeping structures in non-volatile RAM and periodically writes them to the disk. Note that inbuilt non-volatile memory has been quite common in high-end storage devices for a while, and recently it is being used even for regular hard drives [18].

3.1.2 Block Allocation

DHIS performs free-space management at the firmware level, thereby freeing higher-level applications from maintaining information solely for placement of data on disk. Block allocation is done using an explicit `alloc_block` disk primitive. This is important for two reasons. First, higher-level software is unaware of internal disk characteristics and hence cannot make correct decisions about block locality especially when the storage system has a complex hierarchy of disk media internally. For example, an Ext2 file system's allocation algorithm assumes that blocks whose logical block numbers are contiguous are physically contiguous as well. This may not be true in a hierarchical storage system. Second, by managing free-space on disk, DHIS can exploit its knowledge of block-liveness to proactively perform operations such as aggressive replication of hot read-only data, to improve performance and reliability. The block allocation API optionally takes a hint block number to allocate the new block closer to it.

One of the main design goals of DHIS is to enable placement of data blocks at the right RAID-level based on higher level data characteristics such as access patterns, relative importance, etc. Therefore, whenever a block is allocated by the higher-level, the disk has to assign a logical block number for it in the global block namespace, and then allocate a physical block in one of the RAID devices. To enable this, DHIS maintains an allocation bitmap for the logical namespace and separate bitmaps for every underlying physical device. The block-allocation primitive performs two steps: one to allocate a logical block number and the second for a physical block number in one of the lower disks. DHIS adds a `TTABLE` entry whenever a new block is allocated.

3.1.3 Pointer-Based Optimizations

Like a TSD (described in Section 2.2), DHIS includes two disk primitives, one to create logical pointers between blocks called `CREATE_PTR(srcblk,destblk)`, and the other to delete logical pointers called `DELETE_PTR(srcblk,destblk)`. These primitives can be used by higher-level software to communicate pointer relationship to DHIS . DHIS maintains all pointers with respect to the global logical block namespace, and not the physical

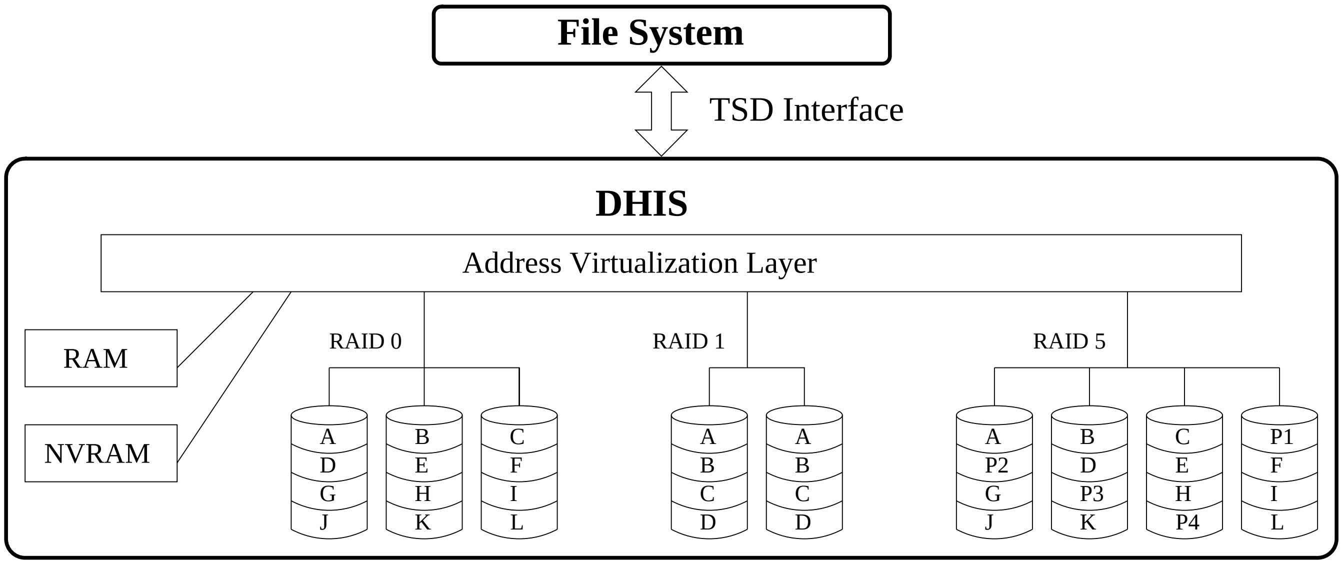


Figure 2: ***DHIS Architecture***

blocks. This allows DHIS to relocate physical blocks transparently without affecting the stored pointer information.

By using pointer knowledge, DHIS performs three key optimizations as follows:

1. All higher-level meta-data blocks (identified as those having outgoing pointers) are placed in the RAID level of highest reliability and best random-access performance. This is because meta-data blocks are more important and accessed more frequently compared to regular data blocks. In our setup, we use RAID1 for this purpose. Note that as the physical destination of blocks are determined at the time of allocation, we do not have information about outgoing pointers for a newly created block and hence we cannot differentiate between data and meta-data for a newly allocated block. Only when the first outgoing pointer is created from a block, DHIS can identify it as a meta-data block. Therefore, DHIS performs dynamic relocation of meta-data blocks to RAID1 as and when the first pointer is created from a block.

2. As meta-data blocks need to be written to disk frequently for reliability reasons, DHIS attempts to absorb the write latency of these blocks by caching writes in NVRAM. As meta-data blocks constitute a small percentage of the total size of storage, NVRAM caching is beneficial. DHIS flushes out the NVRAM contents to RAID1 in configurable periodic intervals of time and also when the device is idle.

3. DHIS exploits its knowledge about block-liveness (differentiating between used and unused blocks) to remove dead blocks (those freed by the file system) from the NVRAM cache and the regular disk cache, for improving the cache utilization.

3.2 Attributes

In this section we describe the set of hints or attributes that higher-level software such as file systems can associate with disk blocks in DHIS. Note that knowledge about pointers at the disk level allows DHIS to inherit attributes of a meta-data block to the sub-tree of data blocks that it points to. For example, to set an attribute for a file, a file system just needs to set an inheritable attribute to the per-file meta-data block, and DHIS automatically inherits the attribute to all blocks belonging to that file.

There have been previous efforts to infer the characteristics of blocks at the disk level without an explicit interface, by using history of accesses [20] or block correlations [13]. However, these methods are quite limited in the range of characteristics they can infer, and often end up being too complex. For example, although it is possible to identify hot and cold blocks using access history, information such as the relative importance of blocks with respect to higher-level applications cannot be inferred easily. Therefore, DHIS provides an explicit interface for communicating a set of well-defined hints or attributes that can be set by higher-level storage software such as file systems.

3.2.1 Attribute Interface

Higher-level software can set attributes using an explicit disk primitive, `DHIS_SETATTR`, by passing a bitmap representing the attributes. Note that attributes in DHIS are normally set to meta-data blocks, and they qualify the characteristics of all blocks in the pointer tree starting from that block. For example, if an Ext2 file system needs to specify the access pattern for a file, it needs to set an appropriate attribute to the corresponding inode block. DHIS automatically groups blocks in the sub-tree and associates the attribute to all such blocks. The following are the attributes that DHIS supports:

- `IMPORTANCE`: determines the relative importance of a data item. Currently DHIS supports this as a Boolean attribute which indicates that an entity is more important than others. This can potentially be extended to support more fine-grained levels based on the diversity in internal storage hardware. Applications, for example, can set this attribute for source files or documents that need to be preserved with the highest level of reliability.

- `ACCESS_PATTERN`: determines if the set of blocks (belonging to the sub-tree of a meta-data block) will be accessed randomly or sequentially. This attribute takes two values: random, or sequential. Applications can set this attribute to files they own based on their access pattern. For example, a simple classification of files based on their types can enable a file system to mark video files as sequential and database index files as random.

- `HOT/COLD`: specifies the frequency in which the particular data item will be accessed. This takes either of these values: hot, and cold. Generally applications can set archival data as cold and frequently updated files such as database write-ahead log files as hot.

- `READ-MOST/WRITE-MOST`: indicates whether a data item will be mostly read or written. For example, binary files such as `/bin/ls` in Unix will be mostly read and will be updated only infrequently. Similarly, file system journals or database log files will predominantly be written.

- `TEMPORARY`: this is a Boolean attribute that indicates whether a data item is temporary (i.e., short-lived) in nature. For example, object files generated by compilers and intermediate files generated by applications such as download managers can be classified as temporary.

The above attributes are the ones that have been currently implemented in the DHIS prototype. In Section 8, we discuss how we plan to extend this set of attributes. Storage software such as file systems can set attributes for appropriate meta-data blocks, using application-specific information. For example, file systems can export an interface to user applications to set attributes at the granularity of files or directories. In such cases, file systems have the responsibility to transform logical abstractions (such as files) into corresponding meta-data blocks and to pass the attributes to DHIS. For example, an Ext2 file system can export an `ioctl` that user applications can use to set attributes to a file identified by a path name. Ext2 can then issue a `DHIS_SETATTR` call with the attribute, for the inode block corresponding to the path name.

3.2.2 *The Ext2DHIS File System*

We developed an attributes-aware file system to support DHIS, as an extended form of the Ext2TSD file system [16]. Ext2TSD is a modified Ext2 file system that supports TSD devices. There are two main differences between a regular Ext2 file system and Ext2TSD. First, Ext2TSD does not perform free-space management, and allocates blocks using the TSD disk API. Second, whenever a new pointer is added or removed for a meta-data block (such as an inode), Ext2TSD issues the corresponding `CREATE_PTR` or `DELETE_PTR` calls to the disk to communicate the pointer.

We have developed Ext2DHIS as an extended Ext2TSD file system that includes an `ioctl` interface for user applications to set attributes to files or directories. Ext2DHIS issues `DHIS_SETATTR` calls to the storage system whenever attributes need to be set or changed. In addition to this, we have developed a simple scheme to set basic attributes automatically for known file name extensions, at the file system level. For example, Ext2DHIS automatically marks files with extensions `.c`, `.cpp`, etc., as important as these may be source files. This provides a simple means to set basic attributes without the need to modify user-level applications.

3.3 Attribute-Based Optimizations

In this section, we describe the optimizations that DHIS achieves using the well-defined set of attributes listed above. First, we present the method we use to choose the right RAID level for a given data item. Second, we describe how better NVRAM utilization can be done by choosing the right candidates to cache. Third, we detail how information about temporary files can aid in reducing disk fragmentation.

3.3.1 *Choosing Optimal RAID Levels*

The three RAID levels that DHIS manages have different performance and reliability characteristics. In this section, we first describe the key characteristics of RAID levels in DHIS and then we detail the policies DHIS adopts to choose the right RAID level to place data.

Characteristics of RAID Levels.

RAID0 performs plain striping across several disks without any redundancy and hence it has the lowest reliability level among the three. However, in terms of performance, RAID0 is good for sequential and random read-write workloads. This is mainly because I/O operations get parallelized across the individual disks when data is striped. In terms of cost per gigabyte, RAID0 is the cheapest as there is no redundancy and the storage capacity is the sum of the individual disk capacities.

RAID1 mirrors data across two or more disks. As the disks contain identical data at all times, data reliability is better as it can tolerate N-1 disk failures where N is the number of mirrored disks. In terms of performance, RAID1 has similar characteristics for both sequential and random I/O. Reads are faster than writes as reads can be parallelized across the N disks. Write speed is in tune with that of a single disk, because for every write, all disks have to be updated, but in parallel. RAID1 has the highest cost per gigabyte as the total capacity of the drives is halved due to mirroring.

RAID5 stripes both data and parity information across three or more drives. In principle it is similar to having a single dedicated parity drive, but parity blocks are distributed across all drives. RAID5 can recover from single disk failures and hence has comparable reliability to a two-disk RAID1. Read performance in RAID5 is similar to that of RAID0. However, for small random writes, RAID5 performs poorly. This is because, for small writes that do not span a complete stripe, computation of new parity involves reading the old contents of the data block and the parity block. In terms of cost per gigabyte, RAID5 is the second best among the three, as there is a single parity block for a stripe.

RAID Placement Policies.

In addition to placing all meta-data blocks in RAID1 (as described in Section 3.1.3), DHIS also adopts placement policies based on higher-level attributes. Table 1 shows the placement policies that DHIS adopts for each combination of attributes. The principles that we use to decide the RAID level for a data item are in tune with the performance and reliability characteristics associated with each RAID level as described above. Note that for data that is `IMPORTANT` and `COLD` we use RAID5 irrespective of its access pattern and read-write characteristics because they are going to be accessed rarely and hence performance is not a significant factor.

3.3.2 *Choosing Candidates for NVRAM Caching*

DHIS chooses candidates for NVRAM caching to maximize the number of absorbed writes through NVRAM. It chooses all meta-data blocks as candidates as described in Section 3.1.3, because meta-data blocks are frequently written and have random access patterns. Similarly, it also chooses blocks with the combination of attributes `HOT`, `WRITE-MOST`, and `RANDOM`, as these are expected to benefit the most from NVRAM caching. We do not choose sequential workloads as candidates and in general they do not benefit much from caching.

DHIS manages NVRAM buffers using a simple mechanism that

IMPORTANT	ACCESS_PATTERN	READ/WRITE-MOST	HOT/COLD	RAID Levels
No	Any	Any	Any	0, 5, 1
Yes	Any	Any	Cold	5, 1, 0
Yes	Not set	Not set	Not set or Hot	5, 1, 0
Yes	Random	Not-set or Write-most	Not set or Hot	1, 5, 0
Yes	Random	Read-most	Not set or Hot	5, 1, 0
Yes	Sequential	Any	Not set or Hot	5, 1, 0

Table 1: ***RAID placement heuristics. The order of RAID levels listed in the last column is the desired order for each combination of attributes. DHIS tries the next level when allocation fails in one of the levels.***

caches writes when the corresponding block is a candidate, and an asynchronous process that flushes NVRAM buffers to disk whenever the disk is idle. When all buffers in the NVRAM are dirty, DHIS passes all subsequent writes to other candidates directly to disk, until NVRAM buffers are flushed out.

3.3.3 Reducing Disk Fragmentation

A fragmented disk can yield poor performance for large files that are accessed sequentially. This is because when there are only fragments of free-space left for allocation, large files may end up spread out across the disk resulting in unnecessary disk seeks. Temporary files that get created and deleted within short intervals of time could exacerbate disk fragmentation, thereby seriously affecting the performance of large files under some scenarios.

DHIS deals with temporary files in a different manner to reduce disk fragmentation. As DHIS is responsible for free-space management, it allocates space for block groups with the TEMPORARY attribute set, in a segregated portion of RAID level 0 (group of blocks at the end of the device). For blocks that are not temporary, DHIS never allocates space from this segregated area. This ensures that temporary files that get created and deleted never interfere with the allocation of regular files, thereby significantly reducing disk fragmentation.

4. IMPLEMENTATION

In this section, we first describe our generic disk functionality prototyping framework, DPROTO, that we built for the Linux kernel 2.6.15. We discuss the implementation details of DHIS over DPROTO.

4.1 DPROTO

We developed DPROTO as a pseudo-device driver that stacks on top of one or more lower-level disk or software RAID drivers, in a single machine. One of the main challenges in developing DPROTO is isolating the resources consumed by components that are supposed to go inside the disk firmware if it were a real implementation. For example, if the functionality being prototyped is a disk-level data compression technique, the part of DPROTO that performs compression has to consume resources that are completely isolated from that used by applications and file systems, which is difficult in a single machine setup.

While developing DPROTO we aimed at isolating key resources, CPU and memory, between disk-level functionality and higher-level applications. For CPU isolation, we use a multiprocessor setup and ensure that disk-level functionality always gets executed in an isolated processor. For memory isolation, we implemented an isolated preallocated memory pool and ensured that disk functionality never accesses memory beyond the preallocated range.

Figure 3 shows the architecture of DPROTO. We implemented the pseudo-device driver as two layers, the upper layer running in the context of the file system, and the lower layer running as a separate thread bound to an isolated CPU. Disk I/O requests generated from the file system reach the upper layer of DPROTO, which adds the request to a shared queue. The lower layer services requests from the queue and eventually passes it down to physical storage. Any disk-level functionality such as compression would be handled by the lower-level service thread and hence runs in an isolated CPU. All memory allocations done by both layers of DPROTO use the preallocated memory pool. Therefore, DPROTO requires specifying the total memory requirement for a given functionality before hand.

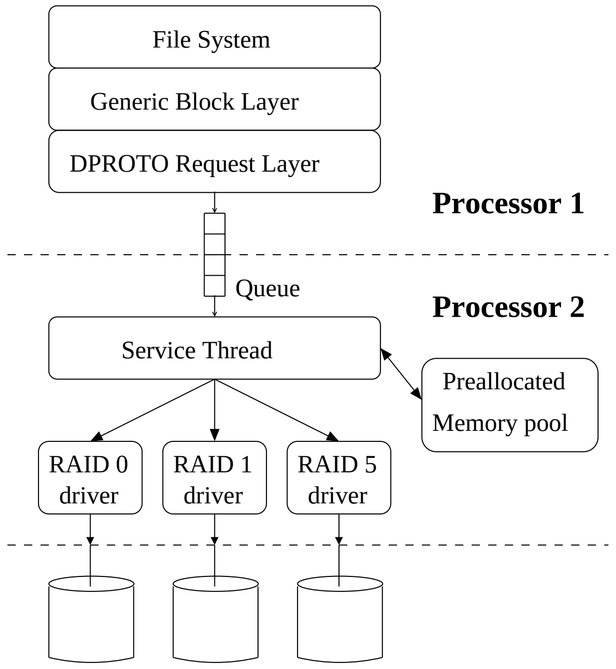


Figure 3: ***DPROTO Architecture***

To test the performance of a disk-level functionality prototyped using DPROTO, the comparison reference can be run with one processor disabled and with the appropriate size of memory preallocated. For example, if a compression disk system is compared to a regular disk system for a particular workload, the regular disk run of the workload has to be done with one processor disabled and the preallocated memory equal to the memory requirement of the compression disk. With this procedure, the comparison becomes fair and closely represents the results of a real implementation.

Our implementation of DPROTO had 5,790 lines of new kernel code and 350 lines of user-level code.

4.2 DHIS Prototype

We implemented a prototype of DHIS using our DPROTO framework. We preallocated the size of each of our data-structures, TTABLE, NVRAM cache, allocation bitmaps, attribute and pointer management structures, and request queue, as a function of the total storage capacity. For the three RAID levels, we stacked DPROTO on top of the regular Linux software RAID drivers for RAID0, RAID1, and RAID5. Our prototype of DHIS had 2,150 lines of kernel code in addition to DPROTO.

5. EVALUATION

We evaluated the performance of DPROTO and our prototype implementation of DHIS to get an estimate of the benefits achieved by attribute-based RAID placement and NVRAM caching. In Section 5.1, we present our evaluation setup. In Section 5.2, we describe the benchmarks that we used. In Section 5.3, we discuss the performance characteristics of our prototyping framework, DPROTO. Finally, in Section 5.4, we show evaluation results for DHIS's RAID placement for several micro-benchmarks, an OLTP workload and a Kernel Compile workload. We also present evaluation results for DHIS's NVRAM caching mechanisms.

5.1 Evaluation Setup

For all benchmarks, we used a 2.8GHz Xeon machine with 1GB RAM, and 6 Maxtor SCSI disks with capacities of 250 GB each, rotational speed of 7,200 RPM and with 8MB cache. We used Fedora Core 6, running a vanilla 2.6.15 kernel.

To ensure a cold cache between benchmark runs, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it.

We observed disk statistics from /proc/diskstats for each of our benchmarks and used it to analyze the reasons behind our results. Disk statistics provide the following information observed by the disk for each benchmark we ran: number of read I/O requests (rio), number of write I/O requests (wio), number of sectors read (rsect), number of sectors written (wsect), number of read requests merged (rmerge), number of write requests merged (wmerge), total time taken for read requests (ruse), and the total time taken for write requests (wuse).

5.2 Benchmarks and Configurations

We used the following benchmarks: Postmark [19], a series of micro-benchmarks, FileBench [1] and Kernel Compile. We discuss each of them below.

We used Postmark [19], a popular file system benchmarking tool, to test the performance of our prototypes. Postmark is I/O-intensive and stresses the file system by creating a large number of small files and then performing a series of file system operations such as directory lookups, creations, and deletions on them. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. Postmark mostly generates a combination of small random reads and writes, and hence we use this for testing the performance of our implementations, under random workloads. The working set of a Postmark benchmark is determined by the number of files to be created initially, and their size range. For all runs of Postmark, we used file sizes ranging from 400KB to 600KB with the base number of files set to 3,000. We chose these parameters in order to create a reasonably large working set for the test machine (1GB RAM). We have mentioned the exact configuration of Postmark used for each test, along with the respective test results.

We also ran a series of micro-benchmarks to test the characteristics that Postmark does not cover. For example, Postmark does not evaluate sequential I/O performance and overheads for large file workloads. Micro-benchmarks also isolate the overheads for specific operations, and hence give a clearer picture of the overheads. We developed a user-level tool that generates one of the following workloads: random read, random write, sequential read, and sequential write. For all runs, we used 4KB read or writes on a single 1.5GB file. For the sequential benchmarks (read and write), we performed sequential 4K I/O on the 1.5GB file 5 times, totalling to 7.5GB of I/O. For random read and write benchmarks, we performed 20,000 and 150,000 4K I/O respectively.

We used FileBench [1] to emulate an OLTP application. Finally, we used the linux kernel 2.6.28 sources for the Kernel Compile workload. We mention more details on the test setup when we discuss the benchmark results for the OLTP and Kernel Compile workloads in Section 5.4.2 and Section 5.4.3 respectively.

5.3 DPROTO Overheads

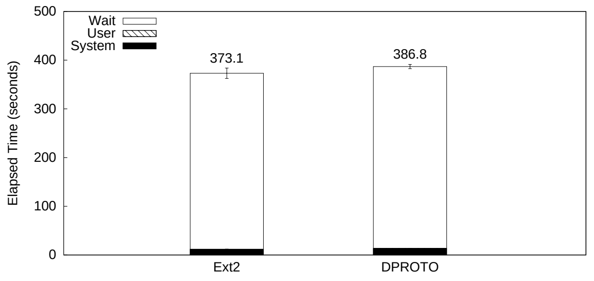


Figure 4: ***Postmark results for DPROTO vs. a regular disk***

We evaluated the performance of DPROTO framework as a null layer that stacks on top of a regular disk. We ran Postmark for two different configurations on an Ext2 file system mounted on the null DPROTO layer, and compared it with Postmark run on a regular disk. Figure 4 shows the overheads of DPROTO compared to a regular disk. The overall elapsed time overhead of DPROTO was only 3.6% compared to regular disks. This is contributed mostly by an increase in wait time, due to an additional level of indirection in the DPROTO request service queue.

5.4 DHIS Results

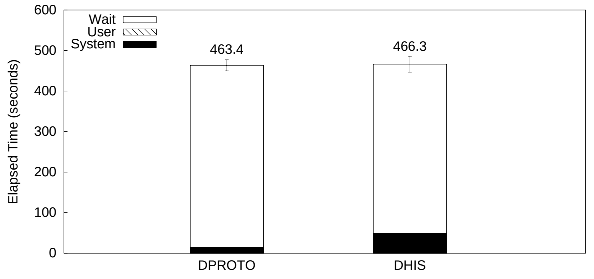


Figure 5: ***Postmark results for Ext2DHIS over DHIS compared to Ext2 over plain DPROTO***

We evaluated the performance of our prototype implementation of DHIS and our optimizations for RAID placement and NVRAM caching.

Figure 5 shows the overheads of DHIS over regular DPROTO. We configured DPROTO to preallocate the same amount of memory that DHIS required for storing its data-structures (128MB). Although the elapsed times for both runs are similar, DHIS has higher system time (13 secs vs. 49 secs) and lower wait time (449 secs vs. 416 secs) compared to regular DPROTO. The system time increase is due to two reasons. First, Ext2DHIS issues ioctls to the pseudo-device driver to communicate pointer information, contributing the major component of system time. Second, the shared queue is protected by a spin lock and hence minor contention causes a busy wait resulting in increased system time. The reduced wait time is because of better spatial locality caused by the disk-level block allocation scheme used by DHIS (compared to file-system–level allocation in Ext2). The scheme co-locates blocks in a greedy fashion without taking into account future file growth.

5.4.1 RAID Placement Optimizations

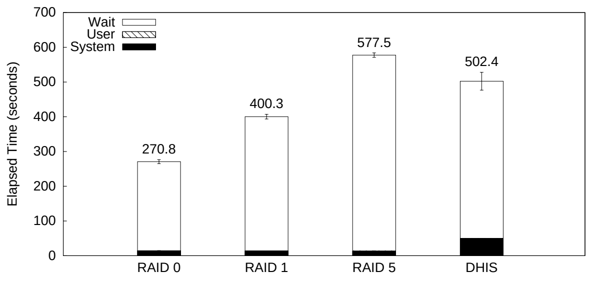


***Figure 6:* Postmark results for Ext2DHIS over DHIS compared to Ext2 over plain DPROTO on RAID1 and RAID5**

To evaluate the benefits of the RAID placement optimizations performed by DHIS, we used Postmark and micro-benchmarks. For all benchmarks, we observed the time taken for the workload on regular DPROTO stacked over individual RAID1 and RAID5 devices and compared them with DHIS. While running the workload over DHIS, we set the `IMPORTANT` and `ACCESS_PATTERN` attributes, so that DHIS would place them in the optimal RAID level.

As Postmark generates mostly a random workload, we ran it with the `RANDOM` attribute set. For micro-benchmarks, we set the `SEQUENTIAL` and `RANDOM` attributes for sequential and random reads and writes respectively.

Figure 6 shows the Postmark results for DHIS as compared to DPROTO on individual RAID1 and RAID5. As evident from the figure, DHIS performs closer to regular RAID1 as it placed the Postmark working set on its RAID1 hierarchy. DHIS has an elapsed time overhead of 25% compared to regular RAID1 although DHIS places all data on RAID1 for this benchmark. This is due to two reasons. First, Postmark creates and deletes a large number of files and hence results in a large amount of pointer operations and attribute updates. This results in increased system time (13 secs vs. 49 secs) as seen from the figure. Second, as pointer operations are synchronous in nature, they block until the DPROTO service thread handles them. This results in increased wait time (386 secs vs. 452 secs). The overheads are more pronounced for the Postmark workload because Postmark is an extreme case of an I/O-intensive workload. In most common workloads, DHIS performs much closer to RAID1 for random workloads (as shown in the micro-benchmark results below).

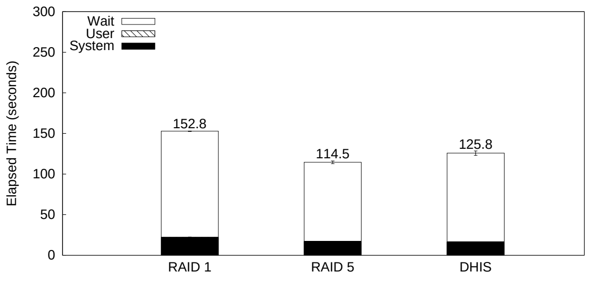
Figure 7 shows the micro-benchmark results for RAID placement. As shown in the graphs, under all cases, DHIS performs close to the fastest of the two RAID levels. Note that for the sequential write workload, DHIS performs 16% better than RAID5. This is because DHIS places all meta-data blocks in RAID1 for maximizing reliability and better performance (as meta-data blocks will mostly be accessed at random). By placing meta-data blocks in RAID1, DHIS has better sequential write characteristics, as random meta-data updates (such as updating the inode) gets absorbed by RAID1 while writing to a large sequential file on RAID5.

5.4.2 OLTP Workload

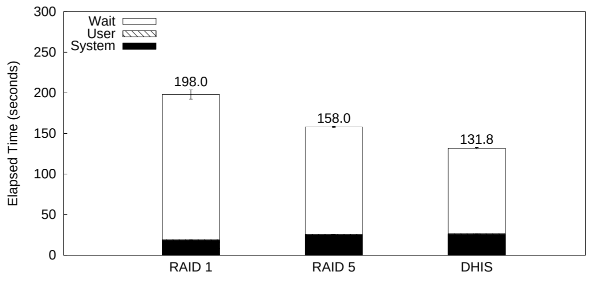
FileBench [1] is a framework for file system workloads. It uses a high level workload language to model the I/O behavior and other characteristics of desired applications. We used FileBench to emulate an OLTP workload. This workload performs transactions on a file system using an I/O model from Oracle 9i. This workload tests for the performance of small random reads and writes to data files and synchronous writes to a log file. It launches a configurable number of reader processes, ten writer processes for asynchronous writing, and a log writer process. The emulation includes the use of Intimate Shared Memory (ISM). ISM is a special kind of shared memory used by DBMS vendors to maximize I/O performance. Because the writes to data files are asynchronous, the throughput is limited mostly by the read performance. Figure 8(a) shows the I/O throughput achieved using DHIS for a varying number of reader processes. The figure also compares these throughput values with those achieved using DPROTO over RAID0, RAID1 and RAID5. The number of reader processes is varied between 25 and 100 in increments of 25 with 10 asynchronous writer processes and a log writer process. The working set for the workload consists of 10 data files each of size 250 MB and a log file of size 250 MB. These file sizes ensure that we have a reasonably large working set for the test machine (1 GB RAM). While benchmarking DHIS, we ran FileBench with the attributes `IMPORTANT` and `READ-MOST` and `ACCESS-PATTERN` set to `RANDOM` on the data files. We observe that DHIS performs close to RAID5 as it chooses RAID5 for the said category of data files. RAID0 has higher throughput relative to RAID1, RAID5, and DHIS for this workload because of the high concurrency in disk accesses but is not applicable for important data. Figure 8(b) shows the latency per operation using the same configurations as above. Again, DHIS performs close to RAID5. Further, the relatively higher rate of increase in latency values for RAID1 with increasing number of reader processes shows that RAID0, RAID5, and DHIS handle I/O concurrency much better than RAID1 for this workload. In summary, setting appropriate attributes on data files for the OLTP workload enables DHIS to make the most efficient data placement decision and it indeed performs close to RAID5.

5.4.3 Kernel Compile Workload

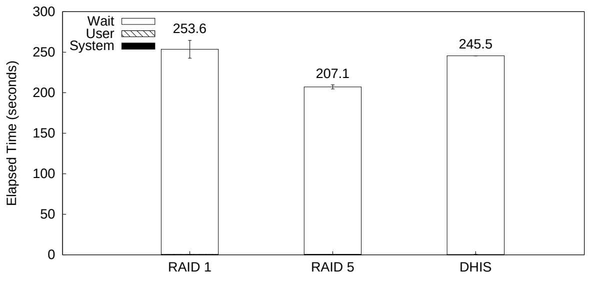
To evaluate the benefits of DHIS when upper layers such as file systems or applications set attributes on files based on file types, we ran a kernel compile workload for four cases. For each case, the sources directory is based on RAID1 storage that is separate from the storage for the build directory. The build directory was configured to be based on one of RAID0, RAID1, RAID5, or DHIS. The upper layer (in this case, the ext2DHIS file system) sets the attribute `TEMPORARY` on binary files. DHIS processes the attribute and places all binary files in RAID0 storage. Figure 9 shows the total number of sectors written in each case. While RAID1 and RAID5 write 82% and 48% more sectors than DHIS, DHIS writes 7% more sectors than RAID0. The overhead of DHIS relative to RAID0 is explained by the fact that DHIS places all metadata blocks in RAID1. Figure 10 shows that all four cases compare



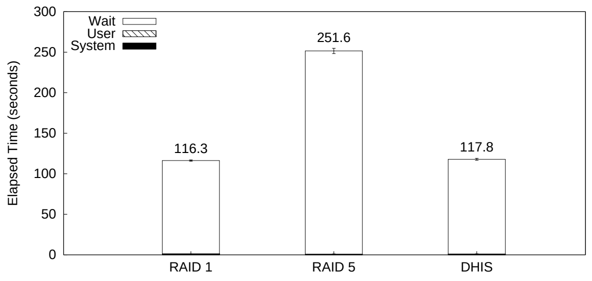
(a) Sequential Read Benchmark



(b) Sequential Write Benchmark

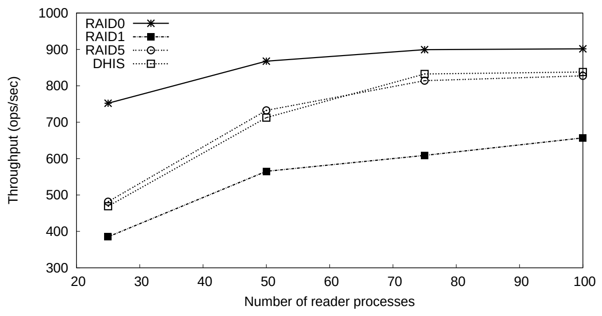


(c) Random Read Benchmark



(d) Random Write Benchmark

Figure 7: ***Microbenchmark results for DHIS. For each benchmark we show the time taken for regular DPROTO directly over RAID1 and RAID5, and compared them with DHIS with access-pattern attributes***



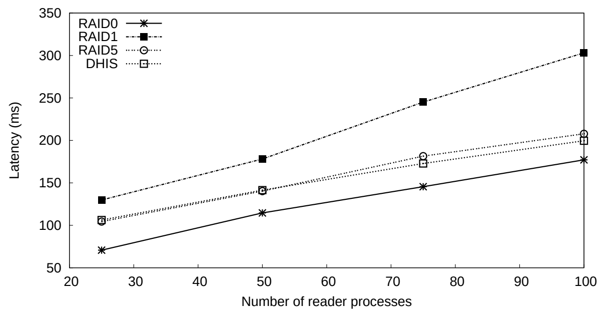


Figure 8: Throughput and Latency values for RAID0, RAID1, RAID5 and DHIS for the FileBench OLTP workload

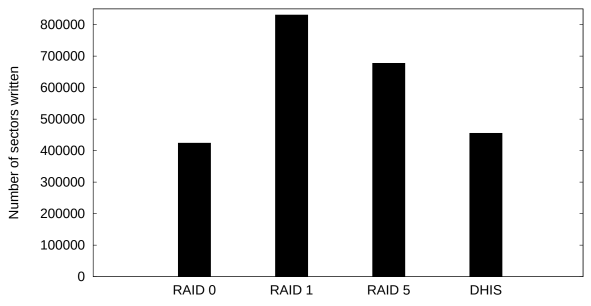


Figure 9: ***Number of sectors written for RAID0, RAID1, RAID5, and DHIS for the Kernel Compile Workload***

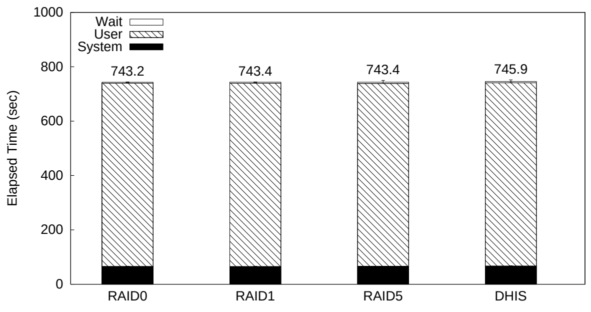


Figure 10: ***Kernel Compile results for RAID0, RAID1, RAID5, and DHIS***

almost equally well in terms of performance. Infact, DHIS has a system time overhead of about 2% because of pointer operations and attribute updates. The lack of apparent performance benefits is because a kernel compile is a CPU-intensive workload. However, other benefits such as power savings and reduction in backup overhead are apparent if we consider the fact that relatively less I/O is being performed and the possibility that RAID0 storage need not be backed up.

5.4.4 NVRAM Caching

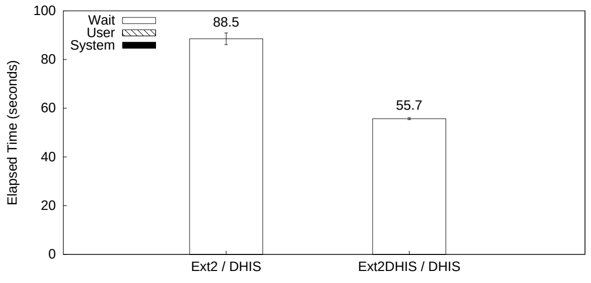


Figure 11: ***Postmark results for Ext2DHIS over DHIS with selective NVRAM caching enabled (right) compared to Ext2 over DHIS (left)***

To evaluate the benefits of caching selected candidates in NVRAM, we compared ext2 over DHIS with ext2DHIS over DHIS. When running ext2DHIS over DHIS, we enabled selective NVRAM caching for DHIS. In both the cases, the file systems were mounted in synchronous mode. Using the pointer information exported by Ext2DHIS, DHIS chooses all meta-data blocks as candidates for NVRAM caching and hence for a synchronous workload most of the meta-data block writes will be absorbed by the NVRAM. Figure 11 shows the benefits of selective NVRAM caching. For this run, we configured Postmark to create 1,000 files with sizes ranging from 10KB to 20KB, and 2,000 operations. We used this smaller configuration as we ran this workload with a synchronous mount of the file systems. As seen from the figure, caching meta-data selectively in NVRAM can improve write performance significantly (37%) for random I/O-intensive workloads.

In summary, we have shown through several benchmarks and workloads that by setting appropriate attributes about data usage, access patterns and importance, DHIS can be made to perform substantially better than traditional storage systems that place data without the knowledge of such attributes.

6. RELATED WORK

Our work builds on the work on type-safe disks by Sivathanu et al. [16] which first proposed the notion of communicating information on logical pointers to the disk system, thus enabling the disk to know about the higher level structure of data. Like type-safe disks, Object-Based Storage devices (OSD) [14] also provide a richer device interface, improving device intelligence. OSDs support the notion of attributes on objects through which higher-level software can communicate object properties to the storage devices. Thus, a hierarchical storage system like DHIS can also be built on top of OSDs.

DHIS can help automate storage administration as envisaged by Self-* Storage [8]. That work proposes the notion of supervisors, workers, and routers for automated administration. Workers are responsible for storage allocations based on observed workloads. Although the Self-* storage architecture would work with workers as block stores, they can work better with intelligent storage systems like DHIS.

Karma [22] provides for an informed management policy in the context of multilevel caches. Like DHIS, Karma leverages application hints to make informed allocation decisions. Whereas Karma focuses on improving cache hit rate, DHIS enables RAID levels to occupy optimal positions in the storage hierarchy depending on data attributes.

The trade-offs between various RAID layout policies in large storage systems are well understood. Therefore, various approaches have been explored to tune these policies based on data-access patterns. One of the earliest systems that sought to address this issue is Hy's AutoRAID [20]. AutoRAID manages two RAID levels: RAID-1 and RAID-5. Newly written data is first placed in RAID-1 and then is slowly migrated into RAID-5 as the data gets cold. One problem with AutoRAID is that this migration cost is paid in the common case, since by default all data starts off in RAID-1. Second, the placement in the right RAID level is based on what the system infers to be the access pattern. This can be quite hard to infer accurately when the workload consists of various independent interleaved streams of access. In contrast, DHIS exploits explicit hints from the higher layers to enable more accurate placement. Also, while AutoRAID only addresses one dimension in this optimization space (namely, choosing RAID levels based on whether data is hot or cold), we have shown in this paper that there are various other attributes to be considered while deciding on an efficient layout and caching strategy.

Another approach that has been explored to address the problem of choosing the right RAID policies is to export information from the RAID system to the higher layers. ExRAID [6] is an example of a system in this category. By exposing fault boundaries and redundancy information to the file system, ExRAID allowed the file system to tune its placement to match its expectations on the characteristics of the data. RAID systems in the industry also adopt a somewhat similar strategy where the volume manager enables creating multiple volumes per RAID level. One could imagine that a file system or database system could then implement custom policies by laying out data in the right volume. RAIF [12] is one such approach. A problem with that approach is that it requires the higher levels to understand the specifics of the range of policies and mechanisms the storage system supports. Researchers have also tried to automate this volume configuration based on offline trace analysis on the workload [2, 3]. However, given the increasing complexity of storage systems and the prevalence of a wide variety of policies for storage layout, this approach is harder to scale. For example, NetApp systems use a form of double-failure protection called Row-Diagonal parity [5] and there are other implementations of RAID6 [11]. Instead, by abstracting higher level data characteristics through well-defined attributes, we bridge this gap without creating a dependency between the file system and the storage system.

Application-level hints can be processed by runtime libraries [21] over native storage interfaces. However, this architecture requires runtime libraries for individual storage resources and is well-suited for cases where the storage resources are distributed.

More recently, there has been work on automatically inferring knowledge about higher-level operations and data structures by using semantic knowledge about the data [4, 17]. For example, semantic disks are capable of inferring that a particular set of blocks are metadata blocks and thus place those blocks in NVRAM for better write performance [17]. One drawback with the inference approach is its complexity and the difficulty of getting such inference always correct. With the more explicit attributes, DHIS can utilize a wider variety of higher-level information; for instance, the NVRAM caching in DHIS extends beyond just file system metadata, and includes arbitrary user-level data that has access patterns that are likely to benefit from NVRAM caching.

7. CONCLUSIONS

The wide variety of techniques available to manage storage layout and reliability results in a difficult question for both storage vendors and developers of higher layers that interact with the storage system: which of these policies should be chosen? Our contributions to address this problem are as follows. First, we presented a new design choice for making optimal data placement decisions in an online fashion. By making intelligent use of the different attributes on data usage and importance, we have shown that a RAID system can achieve much better efficiency in its layout policies while remaining transparent to higher layers. File systems and other higher level layers simply inform the storage system about what they already know about the data, without worrying about how the storage system would use that information, thus freeing higher layers from having to reason about the internals of the storage system. Second, we used a generic disk functionality prototyping framework DPROTO to implement a hierarchical storage system, DHIS. DPROTO isolates key resources, CPU and memory, between disk-level functionality and host applications enabling effective prototyping and benchmarking. Finally, we have shown through an OLTP macro-benchmark and several micro-benchmarks that DHIS achieves benefits that are close to what can be achieved ideally with optimal data placement.

8. FUTURE WORK

In the future, we plan to extend this work with new attributes and optimizations including more intelligent block placement within a specific RAID level, by using higher level hints about projected file growth and more fine-grained file life-time characteristics. We also plan to implement policies for placement in finer-grained reliability levels in RAID, and consider emerging storage hardware such as flash memory.

9. ACKNOWLEDGMENTS

This work was partially made possible by NSF CAREER EIA-0133589 and NSF CCR-0310493 awards.

10. REFERENCES

[1] A. Wilson. The new and improved FileBench. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008. USENIX Association.

[2] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, 2001.

[3] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 175–188, Monterey, CA, January 2002. USENIX Association.

[4] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*, pages 176–187, Washington, DC, USA, 2004. IEEE Computer Society.

[5] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 1–14, San Francisco, CA, March/April 2004. USENIX Association.

[6] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[7] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems. Product description guide, 1999.

[8] Gregory R. Ganger, John D. Strunk, and Andre J. Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, Carnegie Mellon University, August 2003.

[9] IBM. IBM System Storage DS6800. `http://www-03.ibm.com/systems/storage/disk/ds6000/index.html`, 2007.

[10] IBM. IBM System Storage DS8000 Turbo. `http://www-03.ibm.com/systems/storage/disk/ds8000/index.html`, 2007.

[11] J. S. Plank. The RAID-6 Liberation Codes. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008. USENIX Association.

[12] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok. RAIF: Redundant Array of

Independent Filesystems. In *Proceedings of 24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, pages 199–212, San Diego, CA, September 2007. IEEE.

[13] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 173–186, Berkeley, CA, USA, 2004. USENIX Association.

[14] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41:84–90, August 2003. ieeexplore.ieee.org.

[15] Network Appliance Inc. Network Appliance FAS6000 Series. Product Data Sheet, 2006.

[16] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[17] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[18] Seagate Technology. Momentus 5400 PSD Hybrid Hard Drives. `www.seagate.com/www/en-us/products/laptops/momentus/momentus_5400_psd_hybrid/`, 2007.

[19] VERITAS Software. VERITAS file server edition performance brief: A PostMark 1.11 benchmark comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.

[20] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[21] X. Shen and A. Choudhary and C. Matarazzo and P. Sinha. A Distributed Multi-Storage Resource Architecture and I/O Performance Prediction for Scientific Computing. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.

[22] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-all replacement for a multilevel cache. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 169–184, San Jose, CA, February 2007. USENIX Association.