Differentiated Storage Services

Michael Mesnier, Feng Chen, Tian Luo*

Intel Labs
Intel Corporation
Hillsboro, OR

Jason B. Akers

Storage Technologies Group
Intel Corporation
Hillsboro, OR

ABSTRACT

We propose an I/O classification architecture to close the widening semantic gap between computer systems and storage systems. By *classifying* I/O, a computer system can request that different classes of data be handled with different storage system *policies*. Specifically, when a storage system is first initialized, we assign performance policies to predefined classes, such as the filesystem journal. Then, online, we include a classifier with each I/O command (e.g., SCSI), thereby allowing the storage system to enforce the associated policy for each I/O that it receives.

Our immediate application is caching. We present filesystem prototypes and a database proof-of-concept that classify all disk I/O — with very little modification to the filesystem, database, and operating system. We associate caching policies with various classes (e.g., large files shall be evicted before metadata and small files), and we show that end-to-end file system performance can be improved by over a factor of two, relative to conventional caches like LRU. And caching is simply one of many possible applications. As part of our ongoing work, we are exploring other classes, policies and storage system mechanisms that can be used to improve end-to-end performance, reliability and security.

Categories and Subject Descriptors

D.4 [**Operating Systems**]; D.4.2 [**Storage Management**]: [Storage hierarchies]; D.4.3 [**File Systems Management**]: [File organization]; H.2 [**Database Management**]

General Terms

Classification, quality of service, caching, solid-state storage

1. INTRODUCTION

The block-based storage interface is arguably the most stable interface in computer systems today. Indeed, the primary read/write functionality is quite similar to that used by the first commercial disk drive (IBM RAMAC, 1956). Such stability has allowed computer and storage systems to evolve in an independent yet interoperable manner, but at at a cost – it is difficult for computer systems to optimize for increasingly complex storage system internals, and storage systems do not have the semantic information (e.g., on-disk FS and DB data structures) to optimize independently.

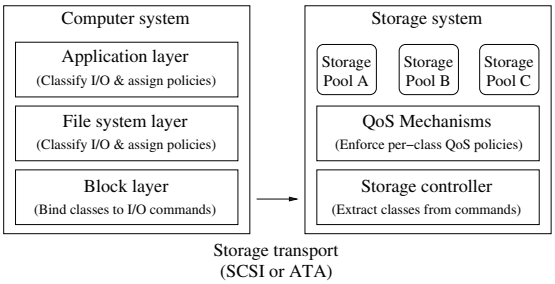


Figure 1: High-level architecture

By way of analogy, shipping companies have long recognized that *classification* is the key to providing differentiated service. Boxes are often classified (kitchen, living room, garage), assigned different policies (deliver-first, overnight, priority, handle-with-care), and thusly treated differently by a shipper (hand-carry, locked van, truck). Separating classification from policy allows customers to pack and classify (label) their boxes once; the handling policies can be assigned on demand, depending on the shipper. And separating policy from mechanism frees customers from managing the internal affairs of the shipper, like which pallets to place their shipments on.

In contrast, modern computer systems expend considerable effort attempting to manage storage system internals, because different classes of data often need different levels of service. As examples, the "middle" of a disk can be used to reduce seek latency, and the "outer tracks" can be used to improve transfer speeds. But, with the increasing complexity of storage systems, these techniques are losing their effectiveness — and storage systems can do very little to help because they lack the semantic information to do so.

We argue that computer and storage systems should operate in the same manner as the shipping industry — by utilizing I/O classification. In turn, this will enable storage systems to enforce per-class QoS policies. See Figure 1.

*The Ohio State University

SOSP '11, October 23-26, 2011, Cascais, Portugal.

Differentiated Storage Services is such a classification framework: I/O is classified in the computer system (e.g., filesystem journal, directory, small file, database log, index, ...), policies are associated with classes (e.g., an FS journal requires low-latency writes, and a database index requires low-latency reads), and mechanisms in the storage system enforce policies (e.g., a cache provides low latency).

Our approach only slightly modifies the existing block interface, so eventual standardization and widespread adoption are practical. Specifically, we modify the OS block layer so that every I/O request carries a classifier. We copy this classifier into the I/O command (e.g., SCSI CDB), and we specify policies on classes through the management interface of the storage system. In this way, a storage system can provide block-level differentiated services (performance, reliability, or security) — and do so on a class-by-class basis. The storage system does not need any knowledge of computer system internals, nor does the computer system need knowledge of storage system internals.

Classifiers describe what the data is, and policies describe how the data is to be managed. Classifiers are handles that the computer system can use to assign policies and, in our SCSI-based prototypes, a classifier is just a number used to distinguish various filesystem classes, like metadata versus data. We also have user-definable classes that, for example, a database can use to classify I/O to specific database structures like an index. Defining the classes (the classification scheme) should be an infrequent operation that happens once for each filesystem or database of interest.

In contrast, we expect that policies will vary across storage systems, and that vendors will differentiate themselves through the policies they offer. As examples, storage system vendors may offer service levels (platinum, gold, silver, bronze), performance levels (bandwidth and latency targets), or relative priority levels (the approach we take in this paper). A computer system must map its classes to the appropriate set of policies, and I/O classification provides a convenient way to do this dynamically when a filesystem or database is created on a new storage system. Table 1 shows a hypothetical mapping of filesystem classes to available performance policies, for three different storage systems.

Beyond performance, there could be numerous other policies that one might associate with a given class, such as replication levels, encryption and integrity policies, perhaps even data retention policies (e.g., secure erase). Rather than attempt to send all of this policy information along with each I/O, we simply send a classifier. This will make efficient use of the limited space in an I/O command (e.g., SCSI has 5 bits that we use as a classifier). In the storage system the classifier can be associated with any number of policies.

We begin with a priority-based performance policy for cache management, specifically for non-volatile caches composed of solid-state drives (SSDs). That is, to each FS and DB class we assign a caching policy (a relative priority level). In practice, we assume that the filesystem or database vendor, perhaps in partnership with the storage system vendor, will provide a default priority assignment that a system administrator may choose to tune.

FS Class	Vendor A: Service levels	Vendor B: Perf. targets	Vendor C: Priorities
Metadata	Platinum	Low lat.	0
Journal	Gold	Low lat.	0
Small file	Silver	Low lat.	1
Large file	Bronze	High BW	2

Table 1: An example showing FS classes mapped to various performance policies. This paper focuses on priorities; lower numbers are higher priority.

We present prototypes for Linux Ext3 and Windows NTFS, where I/O is classified as metadata, journal, directory, or file, and file I/O is further classified by the file size (e.g., ≤4KB ≤16KB, ..., >1GB). We assign a caching priority to each class: metadata, journal, and directory blocks are highest priority, followed by regular file data. For the regular files, we give small files higher priority than large ones.

These priority assignments reflect our goal of reserving cache space for metadata and small files. To this end, we introduce two new block-level caching algorithms: *selective allocation* and *selective eviction*. Selective allocation uses the priority information when allocating I/O in a cache, and selective eviction uses this same information during eviction. The end-to-end performance improvements of selective caching are considerable. Relative to conventional LRU caching, we improve the performance of a file server by 1.8x, an e-mail server by 2x, and metadata-intensive FS utilities (e.g., `find` and `fsck`) by up to 6x. Furthermore, a TCO analysis by Intel IT Research shows that priority-based caching can reduce caching costs by up to 50%, as measured by the acquisition cost of hard drives and SSDs.

It is important to note that in both of our FS prototypes, we do not change which logical blocks are being accessed; we simply classify I/O requests. Our design philosophy is that the computer system continues to see a single logical volume and that the I/O into that volume be classified. In this sense, classes can be considered "hints" to the storage system. Storage systems that know how to interpret the hints can optimize accordingly, otherwise they can be ignored. This makes the solution backward compatible, and therefore suitable for legacy applications.

To further show the flexibility of our approach, we present a proof-of-concept classification scheme for PostgreSQL [33]. Database developers have long recognized the need for intelligent buffer management in the database [10] and in the operating system [45]; buffers are often classified by type (e.g., index vs. table) and access pattern (e.g., random vs. sequential). To share this knowledge with the storage system, we propose a POSIX file flag (`O_CLASSIFIED`). When a file is opened with this flag, the OS extracts classification information from a user-provided data buffer that is sent with each I/O request and, in turn, binds the classifier to the outgoing I/O command. Using this interface, we can easily classify all DB I/O, with only minor modification to the DB and the OS. This same interface can be used by any application. Application-level classes will share the classification space with the filesystem — some of the classifier bits can be reserved for applications, and the rest for the filesystem.

This paper is organized as follows. Section 2 motivates the need for Differentiated Storage Services, highlighting the shortcomings of the block interface and building a case for block-level differentiation. Alternative designs, not based on I/O classification, are discussed. We present our design in Section 3, our FS prototypes and DB proof-of-concept in Section 4, and our evaluation in Section 5. Related work is presented in Section 6, and we conclude in Section 7.

2. BACKGROUND & MOTIVATION

The contemporary challenge motivating Differentiated Storage Services is the integration of SSDs, as caches, into conventional disk-based storage systems. The fundamental limitation imposed by the block layer (lack of semantic information) is what makes effective integration so challenging. Specifically, the block layer abstracts computer systems from the details of the underlying storage system, and *vice versa*.

2.1 Computer system challenges

Computer system performance is often determined by the underlying storage system, so filesystems and databases must be smart in how they allocate on-disk data structures. As examples, the journal (or log) is often allocated in the middle of a disk drive to minimize the average seek distance [37], files are often created close to their parent directories, and file and directory data are allocated contiguously whenever possible. These are all attempts by a computer system to obtain some form differentiated service through intelligent block allocation.

Unfortunately, the increasing complexity of storage systems is making intelligent allocation difficult. Where is the "middle" of the disk, for example, when a filesystem is mounted atop a logical volume with multiple devices, or perhaps a hybrid disk drive composed of NAND and shingled magnetic recording? Or, how do storage system caches influence the latency of individual read/write operations, and how can computer systems reliably manage performance in the context of these caches? One could use models [27, 49, 52] to predict performance, but if the predicted performance is undesirable there is very little a computer system can do to change it.

In general, computer systems have come to expect only best-effort performance from their storage systems. In cases where performance must be guaranteed, dedicated and over-provisioned solutions are deployed.

2.2 Storage system challenges

Storage systems already offer differentiated service, but only at a coarse granularity (logical volumes). Through the management interface of the storage system, administrators can create logical volumes with the desired capacity, reliability, and performance characteristics — by appropriately configuring RAID and caching.

However, before an I/O enters the storage system, valuable semantic information is stripped away at the OS block layer, such as user, group, application, and process information. And, any information regarding on-disk structures is obfuscated. This means that all I/O receives the same treatment within the logical volume.

For a storage system to provide any meaningful optimization within a volume, it must have semantic computer system information. Without help from the computer system, this can be very difficult to get. Consider, for example, that a filename could influence how a file is cached [26], and what would be required for a storage system to simply determine the the name of a file associated with a particular I/O. Not only would the storage system need to understand the on-disk metadata structures of the filesystem, particularly the format of directories and their filenames, but it would have to track all I/O requests that modify these structures. This would be an extremely difficult and potentially fragile process. Expecting storage systems to retain sufficient and up-to-date knowledge of the on-disk structures for each of its attached computer systems may not be practical, or even possible, to realize in practice.

2.3 Attempted solutions & shortcomings

Three schools of thought have emerged to better optimize the I/O between a computer and storage system. Some show that computer systems can obtain more knowledge of storage system internals and use this information to guide block allocation [11, 38]. In some cases, this means managing different storage volumes [36], often foregoing storage system services like RAID and caching. Others show that storage systems can discover more about on-disk data structures and optimize I/O accesses to these structures [9, 41, 42, 43]. Still others show that the I/O interface can evolve and become more expressive; object-based storage and type-safe disks fall into this category [28, 40, 58].

Unfortunately, none of these approaches has gained significant traction in the industry. First, increasing storage system complexity is making it difficult for computer systems to reliably gather information about internal storage structure. Second, increasing computer system complexity (e.g., virtualization, new filesystems) is creating a moving target for semantically-aware storage systems that learn about on-disk data structures. And third, although a more expressive interface could address many of these issues, our industry has developed around a block-based interface, for better or for worse. In particular, filesystem and database vendors have a considerable amount of intellectual property in how blocks are managed and would prefer to keep this functionality in software, rather than offload to the storage system through a new interface.

When a new technology like solid-state storage emerges, computer system vendors prefer to innovate above the block level, and storage system vendors below. But, this tug-of-war has no winner as far as applications are concerned, because considerable optimization is left on the table.

We believe that a new approach is needed. Rather than teach computer systems about storage system internals, or *vice versa*, we can have them agree on shared, block-level goals — and do so through the existing storage interfaces (SCSI and ATA). This will not introduce a disruptive change in the computer and storage systems ecosystem, thereby allowing computer system vendors to innovate above the block level, and storage system vendors below. To accomplish this, we require a means by which block-level goals can be communicated with each I/O request.

3. DESIGN

Differentiated Storage Services closes the semantic gap between computer and storage systems, but does so in a way that is practical in an industry built around blocks. The problem is not the block interface, *per se*, but a lack of information as to how disk blocks are being used.

We must careful, though, to not give a storage system too much information, as this could break interoperability. So, we simply *classify* I/O requests and communicate block-level goals (policies) for each class. This allows storage systems to provide meaningful levels of differentiation, without requiring that detailed semantic information be shared.

3.1 Operating system requirements

We associate a classifier with every block I/O request in the OS. In UNIX and Windows, we add a classification field to the OS data structure for block I/O (the Linux "BIO," and the Windows "IRP") and we copy this field into the actual I/O command (SCSI or ATA) before it is sent to the storage system. The expressiveness of this field is only limited by its size, and in Section 4 we present a SCSI prototype where a 5-bit SCSI field can classify I/O in up to 32 ways.

In addition to adding the classifier, we modify the OS I/O scheduler, which is responsible for coalescing contiguous I/O requests, so that requests with different classifiers are never coalesced. Otherwise, classification information would be lost when two contiguous requests with different classifiers are combined. This does reduce a scheduler's ability to coalesce I/O, but the benefits gained from providing differentiated service to the uncoalesced requests justify the cost, and we quantify these benefits in Section 5.

The OS changes needed to enable filesystem I/O classification are minor. In Linux, we have a small kernel patch. In Windows, we use closed-source filter drivers to provide the same functionality. Section 4 details these changes.

3.2 Filesystem requirements

First, a filesystem must have a *classification scheme* for its I/O, and this is to be designed by a developer that has a good understanding of the on-disk FS data structures and their performance requirements. Classes should represent blocks with similar goals (e.g., journal blocks, directory blocks, or file blocks); each class has a unique ID. In Section 4, we present our prototype classification schemes for Linux Ext3 and Windows NTFS.

Then, the filesystem developer assigns a policy to each class; refer back to the hypothetical examples given in Table 1. How this policy information is communicated to the storage system can be vendor specific, such as through an administrative GUI, or even standardized. The Storage Management Initiative Specification (SMI-S) is one possible avenue for this type of standardization [3]. As a reference policy, also presented in Section 4, we use a priority-based performance policy for storage system cache management.

Once mounted, the filesystem classifies I/O as per the classification scheme. And blocks may be reclassified over time. Indeed, block reuse in the filesystem (e.g., file deletion or defragmentation) may result in frequent reclassification.

3.3 Storage system requirements

Upon receipt of a classified I/O, the storage system must extract the classifier, lookup the policy associated with the class, and enforce the policy using any of its internal mechanisms; legacy systems without differentiated service can ignore the classifier. The mechanisms used to enforce a policy are completely vendor specific, and in Section 4 we present our prototype mechanism (priority-based caching) that enforces the FS-specified performance priorities.

Because each I/O carries a classifier, the storage system does not need to record the class of each block. Once allocated from a particular storage pool, the storage system is free to discard the classification information. So, in this respect, Differentiated Storage Services is a stateless protocol. However, if the storage system wishes to later move blocks across storage pools, or otherwise change their QoS, it must do so in an informed manner. This must be considered, for example, during de-duplication. Blocks from the same allocation pool (hence, same QoS) can be de-duplicated. Blocks from different pools cannot.

If the classification of a block changes due to block re-use in the filesystem, the storage system must reflect that change internally. In some cases, this may mean moving one or more blocks across storage pools. In the case of our cache prototype, a classification change can result in cache allocation, or the eviction of previously cached blocks.

3.4 Application requirements

Applications can also benefit from I/O classification; two good examples are databases and virtual machines. To allow for this, we propose a new file flag `O_CLASSIFIED`. When a file is opened with this flag, we overload the POSIX scatter/gather operations (`readv` and `writev`) to include one extra list element. This extra element points to a 1-byte user buffer that contains the classification ID of the I/O request. Applications not using scatter/gather I/O can easily convert each I/O to a 2-element scatter/gather list. Applications already issuing scatter/gather need only create the additional element.

Next, we modify the OS virtual file system (VFS) in order to extract this classifier from each `readv()` and `writev()` request. Within the VFS, we know to inspect the file flags when processing each scatter/gather operation. If a file handle has the `O_CLASSIFIED` flag set, we extract the I/O classifier and reduce the scatter/gather list by one element. The classifier is then bound to the kernel-level I/O request, as described in Section 3.1. Currently, our user-level classifiers override the FS classifiers. If a user-level class is specified on a file I/O, the filesystem classifiers will be ignored.

Without further modification to POSIX, we can now explore various ways of differentiating user-level I/O. In general, any application with complex, yet structured, block relationships [29] may benefit from user-level classification. In this paper, we begin with the database and, in Section 4, present a proof-of-concept classification scheme for PostgreSQL [33]. By simply classifying database I/O requests (e.g., user tables versus indexes), we provide a simple way for storage systems to optimize access to on-disk database structures.

4. IMPLEMENTATION

We present our implementations of Differentiated Storage Services, including two filesystem prototypes (Linux Ext3 and Windows NTFS), one database proof-of-concept (Linux PostgreSQL), and two storage system prototypes (SW RAID and iSCSI). Our storage systems implement a priority-based performance policy, so we map each class to a priority level (refer back to Table 1 for other possibilities). For the FS, the priorities reflect our goal to reduce small random access in the storage system, by giving small files and metadata higher priority than large files. For the DB, we simply demonstrate the flexibility of our approach by assigning caching policies to common data structures (indexes, tables, and logs).

4.1 OS changes needed for FS classification

The OS must provide in-kernel filesystems with an interface for classifying each of their I/O requests. In Linux, we do this by adding a new classification field to the FS-visible kernel data structure for disk I/O (`struct buffer_head`). This code fragment illustrates how Ext3 can use this interface to classify the OS disk buffers into which an inode (class 5 in this example) will be read:

```
bh->b_class = 5;     /* classify inode buffer */
submit_bh(READ, bh); /* submit read request   */
```

Once the disk buffers associated with an I/O are classified, the OS block layer has the information needed to classify the block I/O request used to read/write the buffers. Specifically, it is in the implementation of `submit_bh` that the generic block I/O request (the BIO) is generated, so it is here that we copy in the FS classifier:

```
int submit_bh(int rw, struct buffer_head * bh) {
    ...
    bio->bi_class = bh->b_class /* copy in class */
    submit_bio(rw, bio);        /* issue read    */
    ...
    return ret;
}
```

Finally, we copy the classifier once again from the BIO into the 5-bit, vendor-specific *Group Number* field in byte 6 of the SCSI CDB. This one-line change is all that is need to enable classification at the SCSI layer:

```
SCpnt->cmnd[6] = SCpnt->request->bio->bi_class;
```

These 5 bits are included with each WRITE and READ command, and we can fill this field in up to 32 different ways (2^5). An additional 3 reserved bits could also be used to classify data, allowing for up to 256 classifiers (2^8), and there are ways to grow even beyond this if necessary (e.g., other reserved bits, or extended SCSI commands).

In general, adding I/O classification to an existing OS is a matter of tracking an I/O as it proceeds from the filesystem, through the block layer, and down to the device drivers. Whenever I/O requests are copied from one representation to another (e.g., from a buffer head to a BIO, or from a BIO to a SCSI command), we must remember to copy the classifier. Beyond this, the only other minor change is to the I/O scheduler which, as previously mentioned, must be modified so that it only coalesces requests that carry the same classifier.

Block layer	LOC	Change made
bio.h	1	Add classifier
blkdev.h	1	Add classifier
buffer_head.h	13	Add classifier
bio.c	2	Copy classifier
buffer.c	26	Copy classifier
mpage.c	23	Copy classifier
bounce.c	1	Copy classifier
blk-merge.c	28	Merge I/O of same class
direct-io.c	60	Classify file sizes
sd.c	1	Insert classifier into CDB

Table 2: Linux 2.6.34 files modified for I/O classification. Modified lines of code (LOC) shown.

Overall, adding classification to the Linux block layer requires that we modify 10 files (156 lines of code), which results in a small kernel patch. Table 2 summarize the changes. In Windows, the changes are confined to closed-source filter drivers. No kernel code needs to be modified because, unlike Linux, Windows provides a stackable filter driver architecture for intercepting and modifying I/O requests.

4.2 Filesystem prototypes

A filesystem developer must devise a classification scheme and assign storage policies to each class. The goals of the filesystem (performance, reliability, or security) will influence how I/O is classified and policies are assigned.

4.2.1 Reference classification scheme

The classification schemes for the Linux Ext3 and Windows NTFS are similar, so we only present Ext3. Any number of schemes could have been chosen, and we begin with one well-suited to minimizing random disk access in the storage system. The classes include metadata blocks, directory blocks, journal blocks, and regular file blocks. File blocks are further classified by the file size ($\leq$4KB, $\leq$16KB, $\leq$64KB, $\leq$256KB, ..., $\leq$1GB, >1GB) — 11 file size classes in total.

The goal of our classification scheme is to provide the storage system with a way of prioritizing which blocks get cached and the eviction order of cached blocks. Considering the fact that metadata and small files can be responsible for the majority of the disk seeks, we classify I/O in such a way that we can separate these random requests from large-file requests that are commonly accessed sequentially. Database I/O is an obvious exception and, in Section 4.3 we introduce a classification scheme better suited for the database.

Table 3 (first two columns) summarizes our classification scheme for Linux Ext3. Every disk block that is written or read falls into exactly one class. Class 0 (unclassified) occurs when I/O bypasses the Ext3 filesystem. In particular, all I/O created during filesystem creation (`mkfs`) is unclassified, as there is no mounted filesystem to classify the I/O. The next 5 classes (superblocks through indirect data blocks) represent filesystem metadata, as classified by Ext3 after it has been mounted. Note, the unclassified metadata blocks will be re-classified as one of these metadata types when they are first accessed by Ext3. Although we differentiate metadata classes 1 through 5, we could have combined them into one class. For example, it is not critical that we

Ext3 Class	Class ID	Priority
Superblock	1	0
Group Descriptor	2	0
Bitmap	3	0
Inode	4	0
Indirect block	5	0
Directory entry	6	0
Journal entry	7	0
File <= 4KB	8	1
File <= 16KB	9	2
...	...	...
File > 1GB	18	11
Unclassified	0	12

Table 3: Reference classes and caching priorities for Ext3. Each class is assigned a unique SCSI Group Number and assigned a priority (0 is highest).

Ext3	LOC	Change made
balloc.c	2	Classify block bitmaps
dir.c	2	Classify inodes tables
ialloc.c	2	Classify inode bitmaps
inode.c	94	Classify indirect blocks, inodes, dirs, and file sizes
super.c	15	Classify superblocks, journal blocks, and group descriptors
commit.c	3	Classify journal I/O
journal.c	6	Classify journal I/O
revoke.c	2	Classify journal I/O

Table 4: Ext3 changes for Linux 2.6.34.

differentiate superblocks and block bitmaps, as these structures consume very little disk (and cache) space. Still, we do this for illustrative purposes and system debugging.

Continuing, class 6 represents directory blocks, class 7 journal blocks, and 8-18 are the file size classes. File size classes are only approximate. As a file is being created, the file size is changing while writes are being issued to the storage system; files can also be truncated. Subsequent I/O to a file will reclassify the blocks with the latest file size.

Approximate file sizes allow the storage system to differentiate small files from large files. For example, a storage system can cache all files 1MB or smaller, by caching all the file blocks with a classification up to 1MB. The first 1MB of files larger than 1MB may also fall into this category until they are later reclassified. This means that small files will fit entirely in cache, and large files may be partially cached with the remainder stored on disk.

We classify Ext3 using 18 of the 32 available classes from a 5-bit classifier. To implement this classification scheme, we modify 8 Ext3 files (126 lines of code). Table 4 summarizes our changes.

The remaining classes (19 through 31) could be used in other ways by the FS (e.g., text vs. binary, media file, bootable, read-mostly, or hot file), and we are exploring these as part of our future work. The remaining classes could also be used by user-level applications, like the database.

4.2.2 Reference policy assignment

Our prototype storage systems implement 16 priorities; to each class we assign a priority (0 is the highest). Metadata, journal, and directory blocks are highest priority, followed by the regular file blocks. 4KB files are higher priority than 16KB files, and so on. Unclassified I/O, or the unused metadata created during file system creation, is assigned the lowest priority. For this mapping, we only require 13 priorities, so 3 of the priority levels (13-15) are unused. See Table 3.

This priority assignment is specifically tuned for a file server workload (e.g., SPECsfs), as we will show in Section 5, and reflects our bias to optimize the filesystem for small files and metadata. Should this goal change, the priorities could be set differently. Should the storage system offer policies other than priority levels (like those in Table 1), the FS classes would need to be mapped accordingly.

4.3 Database proof-of-concept

In addition to the kernel-level I/O classification interface described in Section 4.1, we provide a POSIX interface for classifying user-level I/O. The interface builds on the scatter/gather functionality already present in POSIX.

Using this new interface, we classify all I/O from the PostgreSQL open source database [33]. As with FS classification, user-level classifiers are just numbers used to distinguish the various I/O classes, and it is the responsibility of the application (a DB in this case) to design a classification scheme and associate storage system policies with each class.

4.3.1 A POSIX interface for classifying DB I/O

We add an additional scatter/gather element to the POSIX `readv` and `writev` system calls. This element points to a user buffer that contains a classifier for the given I/O. To use our interface, a file is opened with the flag `O_CLASSIFIED`. When this flag is set, the OS will assume that all scatter/gather operations contain $1 + n$ elements, where the first element points to a classifier buffer and the remaining n elements point to data buffers. The OS can then extract the classifier buffer, bind the classifier to the kernel-level I/O (as described in Section 4.1), reduce the number of scatter gather elements by one, and send the I/O request down to the filesystem. Table 5 summarizes the changes made to the VFS to implement user-level classification. As with kernel-level classification, this is a small kernel patch.

The following code fragment illustrates the concept for a simple program with a 2-element gathered-write operation:

```
unsigned char class = 23;  /* a class ID */
int fd = open("foo", O_RDWR|O_CLASSIFIED);
struct iovec iov[2];       /* an sg list */

iov[0].iov_base = &class; iov[0].iov_len = 1;
iov[1].iov_base = "Hello, world!";
iov[1].iov_len = strlen("Hello, world!");
rc = writev(fd, iov, 2);   /* 2 elements */
close(fd);
```

The filesystem will classify the file size as described in Section 4.2, but we immediately override this classification with the user-level classification, if it exists. Combining user-level and FS-level classifiers is an interesting area of future work.

OS file	LOC	Change made
filemap.c	50	Extract class from sg list
mm.h	4	Add classifier to readahead
readahead.c	22	Add classifier to readahead
mpage.h	1	Add classifier to page read
mpage.c	5	Add classifier to page read
fs.h	1	Add classifier to FS page read
ext3/inode.c	2	Add classifier to FS page read

Table 5: Linux changes for user-level classification.

DB class	Class ID
Unclassified	0
Transaction Log	19
System table	20
Free space map	21
Temporary table	22
Random user table	23
Sequential user table	24
Index file	25
Reserved	26-31

Table 6: A classification scheme PostgreSQL. Each class is assigned a unique number. This number is copied into the 5-bit SCSI Group Number field in the SCSI WRITE and READ commands.

4.3.2 A DB classification scheme

Our proof-of-concept PostgreSQL classification scheme includes the transaction log, system tables, free space maps, temporary tables, user tables, and indexes. And we further classify the user tables by their access pattern, which the PostgreSQL database already identifies, internally, as random or sequential. Passing this access pattern information to the storage system avoids the need for (laborious) sequential stream detection.

Table 6 summarizes our proof-of-concept DB classification scheme, and Table 7 shows the minor changes required of PostgreSQL. We include this database example to demonstrate the flexibility of our approach and the ability to easily classify user-level I/O. How to properly assign block-level caching priorities for the database is part of our current research, but we do share some early results in Section 5 to demonstrate the performance potential.

4.4 Storage system prototypes

With the introduction of solid-state storage, storage system caches have increased in popularity. Examples include LSI's CacheCade and Adaptec's MaxIQ. Each of these systems use solid-state storage as a persistent disk cache in front of a traditional disk-based RAID array.

We create similar storage system caches and apply the necessary modification to take advantage of I/O classification. In particular, we introduce two new caching algorithms: *selective allocation* and *selective eviction*. These algorithms inspect the relative priority of each I/O and, as such, provide a mechanism by which computer system performance policies can be enforced in a storage system. These caching algorithms build upon a baseline cache, such as LRU.

DB file	LOC	Change made
rel.h	6	Pass classifier to storage manager
xlog.c	7	Classify transaction log
bufmgr.c	17	Classify indexes, system tables, and regular tables
freelist.c	7	Classify sequential vs. random
smgr.c/md.c	21	Assign SCSI groups numbers
fd.c	20	Add classifier to scatter/gather and classify temp. tables

Table 7: PostgreSQL changes.

4.4.1 Our baseline storage system cache

Our baseline cache uses a conventional write-back cache with LRU eviction. Recent research shows that solid-state LRU caching solutions are not cost-effective for enterprise workloads [31]. We confirm this result in our evaluation, but also build upon it by demonstrating that a conventional LRU algorithm *can* be cost-effective with Differentiated Storage Services. Algorithms beyond LRU [13, 25] may produce even better results.

A solid-state drive is used as the cache, and we divide the SSD into a configurable number of allocation units. We use 8 sectors (4KB, a common memory page size) as the allocation unit, and we initialize the cache by contiguously adding all of these allocation units to a *free list*. Initially, this free list contains every sector of the SSD.

For new write requests, we allocate cache entries from this free list. Once allocated, the entries are removed from the free list and added to a *dirty list*. We record the entries allocated to each I/O, by saving the mapping in a hash table keyed by the logical block number.

A *syncer daemon* monitors the size of the free list. When the free list drops below a low watermark, the syncer begins cleaning the dirty list. The dirty list is sorted in LRU order. As dirty entries are read or written, they are moved to the end of the dirty list. In this way, the syncer cleans the least recently used entries first. Dirty entries are read from the SSD and written back to the disk. As entries are cleaned, they are added back to the free list. The free list is also sorted in LRU order, so if clean entries are accessed while in the free list, they are moved to the end of the free list.

It is atop this baseline cache that we implement selective allocation and selective eviction.

4.4.2 Conventional allocation

Two heuristics are commonly used by current storage systems when deciding whether to allocate an I/O request in the cache. These relate to the size of the request and its access pattern (random or sequential). For example, a 256KB request in NTFS tells you that the file the I/O is directed to is at least 256KB in size, and multiple contiguous 256KB requests indicate that the file may be larger. It is the small random requests that benefit most from caching, so large requests or requests that appear to be part of a sequential stream will often bypass the cache, as such requests are just as efficiently served from disk. There are at least two fundamental problems with this approach.

First, the block-level request size is only partially correlated with file size. Small files can be accessed with large requests, and large files can be accessed with small requests. It all depends on the application request size and caching model (e.g., buffered or direct). A classic example of this is the NTFS Master File Table (MFT). This key metadata structure is a large, often sequentially written file. Though when read, the requests are often small and random. If a storage system were to bypass the cache when the MFT is being written, subsequent reads would be forced to go to disk. Fixing this problem would require that the MFT be distinguished from other large files and, without an I/O classification mechanism, this would not be easy.

The second problem is that operating systems have a maximum request size (e.g., 512KB). If one were to make a caching decision based on request size, one could not differentiate file sizes that were larger than this maximum request. This has not been a problem with small DRAM caches, but solid-state caches are considerably larger and can hold many files. So, knowing that a file is, say, 1MB as opposed to 1GB is useful when making a caching decision. For example, it can be better to cache more small files than fewer large ones, which is particularly the case for file servers that are seek-limited from small files and their metadata.

4.4.3 Selective allocation

Because of the above problems, we do not make a cache allocation decision based on request size. Instead, for the FS prototypes, we differentiate metadata from regular files, and we further differentiate the regular files by size.

Metadata and small files are always cached. Large files are conditionally cached. Our current implementation checks to see if the syncer daemon is active (cleaning dirty entries), which indicates cache pressure, and we opt to not cache large files in this case (our configurable cut-off is 1MB or larger — such blocks will bypass the cache). However, an idle syncer daemon indicates that there is space in the cache, so we choose to cache even the largest of files.

4.4.4 Selective eviction

Selective eviction is similar to selective allocation in its use of priority information. Rather than evict entries in strict LRU order, we evict the lowest priority entries first. This is accomplished by maintaining a dirty list for each I/O class. When the number of free cache entries reaches a low watermark, the syncer cleans the lowest priority dirty list first. When that list is exhausted, it selects the next lowest priority list, and so on, until a high watermark of free entries is reached and the syncer is put to sleep.

With selective eviction, we can completely fill the cache without the risk of priority inversion. For an FS, this allows the caching of larger files, but not at the expense of evicting smaller files. Large files will evict themselves under cache pressure, leaving the small files and metadata effectively pinned in the cache. High priority I/O will only be evicted after all lower priority data has been evicted. As we illustrate in our evaluation, small files and metadata rarely get evicted in our enterprise workloads, which contain realistic mixes of small and large file size [29].

4.4.5 Linux implementation

We implement a SW cache as RAID level 9 in the Linux RAID stack (MD).[1] The mapping to RAID is a natural one. RAID levels (e.g., 0, 1, 5) and the nested versions (e.g., 10, 50) simply define a static mapping from logical blocks within a volume to physical blocks on storage devices. RAID-0, for example, specifies that logical blocks will be allocated round-robin. A Differentiated Storage Services architecture, in comparison, provides a dynamic mapping. In our implementation, the classification scheme and associated policies provide a mapping to either the cache device or the storage device, though one might also consider a mapping to multiple cache levels or different storage pools.

Managing the cache as a RAID device allows us to build upon existing RAID management utilities. We use the Linux *mdadm* utility to create a cached volume. One simply specifies the storage device and the caching device (devices in `/dev`), both of which may be another RAID volume. For example, the cache device may be a mirrored pair of SSDs, and the storage device a RAID-50 array. Implementing Differentiated Storage Services in this manner makes for easy integration into existing storage management utilities.

Our SW cache is implemented in a kernel RAID module that is loaded when the cached volume is created; information regarding the classification scheme and priority assignment are passed to the module as runtime parameters. Because the module is part of the kernel, I/O requests are terminated in the block layer and never reach the SCSI layer. The I/O classifiers are, therefore, extracted directly from the block I/O requests (BIOs), not the 5-bit classification field in the SCSI request.

4.4.6 iSCSI implementation

Our second storage system prototype is based on iSCSI [12]. Unlike the RAID-9 prototype, iSCSI is OS-independent and can be accessed by both Linux and Windows. In both cases, the I/O classifier is copied into the SCSI request on the host. On the iSCSI target the I/O classifier is extracted from the request, the priority of the I/O class is determined, and a caching decision is made. The caching implementation is identical to the RAID-9 prototype.

5. EVALUATION

We evaluate our filesystem prototypes using a file server workload (based on SPECsfs [44]), an e-mail server workload (modeled after the Swiss Internet Analysis [30]), a set of filesystem utilities (`find`, `tar`, and `fsck`), and a database workload (TPC-H [47]).

We present data from the Linux RAID-9 implementation for the filesystem workloads; NTFS results using our iSCSI prototype are similar. For Linux TPC-H, we use iSCSI.

5.1 Experimental setup

All experiments are run on a single Linux machine. Our Linux system is a 2-way quad-core Xeon server system (8 cores) with 8GB of RAM. We run Fedora 13 with a 2.6.34 kernel modified as described in Section 4. As such, the Ext3

[1]RAID-9 is not a standard RAID level, but simply a way for us to create cached volumes in Linux MD.

File size	File server	E-mail server
1K	17%	0
2K	16%	24%
4K	16%	26%
8K	7%	18%
16K	7%	12%
32K	9%	6%
64K	7%	5%
128K	5%	3%
256K	5%	2%
512K	4%	2%
1M	3%	1%
2M	2%	0%
8M	1%	0%
10M	0%	1%
32M	1%	0%

Table 8: File size distributions.

filesystem is modified to classify all I/O, the block layer copies the classification into the Linux BIO, and the BIO is consumed by our cache prototype (a kernel module running in the Linux RAID (MD) stack).

Our storage device is a 5-disk LSI RAID-1E array. Atop this base device we configure a cache as described in Section 4.4.5, or 4.4.6 (for TPC-H); an Intel®32GB X25-E SSD is used as the cache. For each of our tests, we configure a cache that is a fraction of the used disk capacity (10-30%).

5.2 Workloads

Our file server workload is based on SPECsfs2008 [44]; the file size distributions are shown in Table 8 (File server). The setup phase creates 262,144 files in 8,738 directories (SFS specifies 30 files per directory). The benchmark performs 262,144 transactions against this file pool, where a transaction is reading an existing file or creating a new file. The read/write ratio is 2:1. The total capacity used by this test is 184GB, and we configure an 18GB cache (10% of the file pool size). We preserve the file pool at the end of the file transactions and run a set of filesystem utilities. Specifically, we search for a non-existent file (`find`), archive the filesystem (`tar`), and then check the filesystem for errors (`fsck`).

Our e-mail server workload is based on a study of e-mail server file sizes [30]. We use a request size of 4KB and a read/write ratio of 2:1. The setup phase creates 1 million files in 1,000 directories. We then perform 1 million transactions (reading or creating an e-mail) against this file pool. The file size distribution for this workload is shown in Table 8 (E-mail server). The total disk capacity used by this test is 204GB, and we configure a 20GB cache.

Finally, we run the TPC-H decision support workload [47] atop our modified PostgreSQL [33] database (Section 4.3). Each PostgreSQL file is opened with the flag `O_CLASSIFIED`, thereby enabling user-level classification and disabling file size classification from Ext3. We build a database with a scale factor of 8, resulting in an on-disk footprint of 29GB, and we run the I/O intensive queries (2, 17, 18, and 19) back-to-back. We compare 8GB LRU and LRU-S caches.

5.3 Test methodology

We use an in-house, file-based workload generator for the file and e-mail server workloads. As input, the generator takes a file size distribution, a request size distribution, a read/write ratio, and the number of subdirectories.

For each workload, our generator creates the specified number of subdirectories and, within these subdirectories, creates files using the specified file and write request size distribution. After the pool is created, transactions are performed against the pool, using these same file and request size distributions. We record the number of files written/read per second and, for each file size, the 95th percentile (worst case) latency, or the time to write or read the entire file.

We compare the performance of three storage configurations: no SSD cache, an LRU cache, and an enhanced LRU cache (LRU-S) that uses selective allocation and selective eviction. For the cached tests, we also record the contents of the cache on a class-by-class basis, the read hit rate, and the eviction overhead (percentage of transferred blocks related to cleaning the cache). These three metrics are performance indicators used to explain the performance differences between LRU and LRU-S. Elapsed time is used as the performance metric in all tests.

5.4 File server

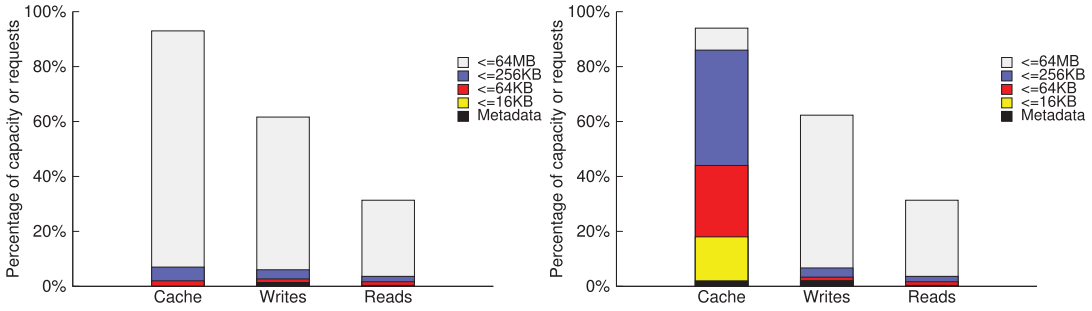
Figure 2a shows the contents of the LRU cache at completion of the benchmark (left bar), the percentage of blocks written (middle bar), and the percentage of blocks read (right bar). The cache bar does not exactly add to 100% due to round-off.[2] Although the cache activity (and contents) will naturally differ across applications, these results are representative for a given benchmark across a range of different cache sizes.

As shown in the figure, the LRU breakdown is similar to the blocks written and read. Most of the blocks belong to large files — a tautology given the file sizes in SPECsfs2008 (most files are small, but most of the data is in large files). Looking again at the leftmost bar, one sees that nearly the entire cache is filled with blocks from large files. The smallest sliver of the graph (bottommost layer of cache bar) represents files up to 64KB in size. Smaller files and metadata consume less than 1% and cannot be seen.

Figure 2b shows the breakdown of the LRU-S cache. The write and read breakdown are identical to Figure 2a, as we are running the same benchmark, but we see a different outcome in terms of cache utilization. Over 40% of the cache is consumed by files 64KB and smaller, and metadata (bottommost layer) is now visible. Unlike LRU eviction alone, selective allocation and selective eviction limit the cache utilization of large files. As utilization increases, large-file blocks are the first to be evicted, thereby preserving small files and metadata.

Figure 3a compares read hit rates. With a 10% cache, the read hit rate is approximately 10%. Given the uniformly random distribution of the SPECsfs2008 workload, this result is expected. However, although the read hit rates are

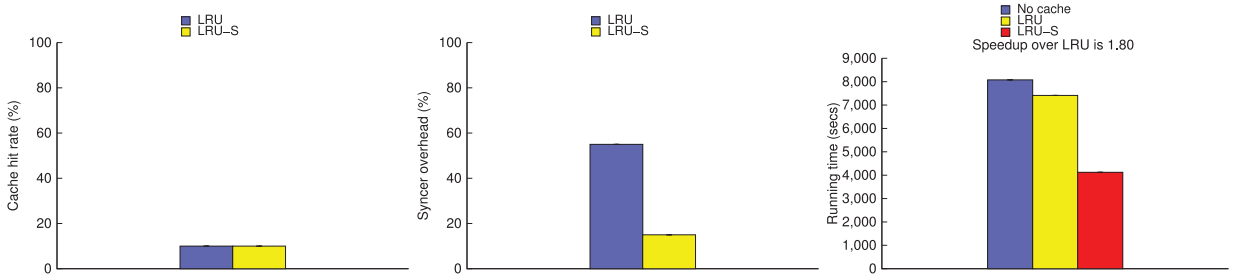
[2]Some of the classes consume less than 1% and round to 0.



(a) LRU cache and I/O breakdown

(b) LRU-S cache and I/O breakdown

Figure 2: SFS results. Cache contents and breakdown of blocks written/read.



(a) Read hit rate

(b) Syncer overhead

(c) SFS performance

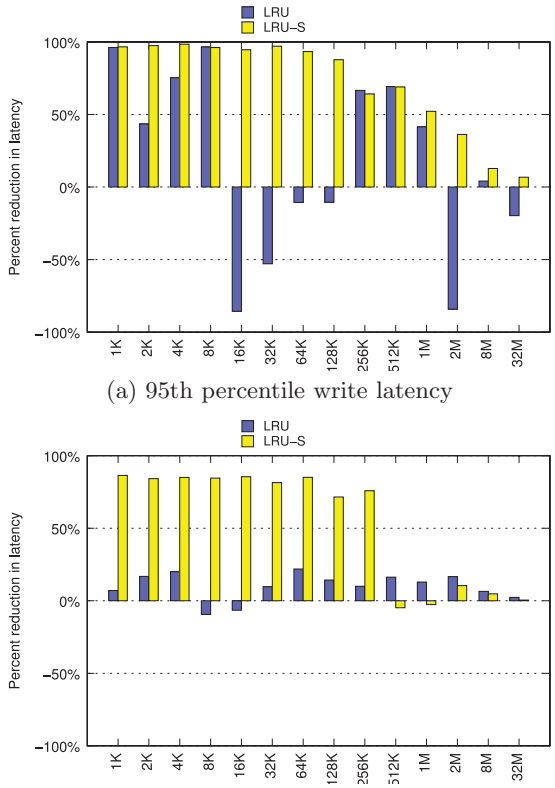
Figure 3: SFS performance indicators

identical, the miss penalties are not. In the case of LRU, most of the hits are to large files. In the case of LRU-S, the hits are to small files and metadata. Given the random seeks associated with small file and metadata, it is better to miss on large sequential files.

Figure 3b compares the overhead of the syncer daemon, where overhead is the percentage of transferred blocks due to cache evictions. When a cache entry is evicted, the syncer must read blocks from the cache device and write them back to the disk device — and this I/O can interfere with application I/O. Selective allocation can reduce the job of the syncer daemon by fencing off large files when there is cache pressure. As a result, we see the percentage of I/O related to evictions drop by more than a factor or 3. This can translate into more available performance for the application workload.

Finally, Figure 3c shows the actual performance of the benchmark. We compare the performance of no cache, an LRU cache, and LRU-S. Performance is measured in running time, so smaller is better. As can be seen in the graph, an LRU cache is only slightly better than no cache at all, and an LRU-S cache is 80% faster than LRU. In terms of running time, the no-cache run completes in 135 minutes, LRU in minutes 124, and LRU-S in 69 minutes.

The large performance difference can also be measured by the improvement in file latencies. Figures 4a and 4b compare the 95th percentile latency of write and read operations, where latency is the time to write or read an entire file. The x-axis represents the file sizes (as per SPECsfs2008) and the y-axis represents the reduction in latency relative to no



(a) 95th percentile write latency

(b) 95th percentile read latency

Figure 4: SFS file latencies

cache at all. Although LRU and LRU-S reduce write latency equally for many of the file sizes (e.g., 1KB, 8KB, 256KB, and 512KB), LRU suffers from outliers that account for the increase in 95th percentile latency. The bars that extend below the x-axis indicate that LRU *increased* write latency relative to no cache, due to cache thrash. And the read latencies show even more improvement with LRU-S. Files 256KB and smaller have latency reductions greater than 50%, compared to the improvements in LRU which are much more modest. Recall, with a 10% cache, only 10% of the working set can be cached. Whereas LRU-S uses this 10% to cache small files and metadata, standard LRU wastes the cache on large, sequentially-accessed files. Stated differently, the cache space we save by evicting large files allows for many more small files to be cached.

5.5 E-mail server

The results from the e-mail server workload are similar to the file server. The read cache hit rate for both LRU and LRU-S is 11%. Again, because the files are accessed with a uniformly random distribution, the hit rate is correlated with the size of the working set that is cached. The miss penalties are again quite different. LRU-S reduces the read latency considerably. In this case, files 32KB and smaller see a large read latency reduction. For example, the read latency for 2KB e-mails is 85ms, LRU reduces this to 21ms, and LRU-S reduces this to 4ms (a reduction of 81% relative to LRU).

As a result of the reduced miss penalty and lower eviction overhead (reduced from 54% to 25%), the e-mail server workload is twice as fast when running with LRU-S. Without any cache, the test completes the 1 million transactions in 341 minutes, LRU completes in 262 minutes, and LRU-S completes in 131 minutes.

Like the file server, an e-mail server is often throughput limited. By giving preference to metadata and small e-mails, significant performance improvements can be realized. This benchmark also demonstrates the flexibility of our FS classification approach. That is, our file size classification is sufficient to handle both file and e-mail server workloads, which have very different file size distributions.

5.6 FS utilities

The FS utilities further demonstrate the advantages of selective caching. Following the file server workload, we search the filesystem for a non-existent file (`find`, a 100% read-only metadata workload), create a tape archive of an SFS subdirectory (`tar`), and check the filesystem (`fsck`).

For the `find` operation, the LRU configuration sees an 80% read hit rate, compared to 100% for LRU-S. As a result, LRU completes the find in 48 seconds, and LRU-S in 13 (a 3.7x speedup). For `tar`, LRU has a 5% read hit rate, compared to 10% for LRU-S. Moreover, nearly 50% of the total I/O for LRU is related to syncer daemon activity, as LRU write-caches the tar file, causing evictions of the existing cache entries and leading to cache thrash. In contrast, the LRU-S fencing algorithm directs the tar file to disk. As as result, LRU-S completes the archive creation in 598 seconds, compared to LRU's 850 seconds (a 42% speedup).

Finally, LRU completes `fsck` in 562 seconds, compared to 94 seconds for LRU-S (a 6x speedup). Unlike LRU, LRU-S retains filesystem metadata in the cache, throughout all of the tests, resulting in a much faster consistency check of the filesystem.

5.7 TPC-H

As one example of how our proof-of-concept DB can prioritize I/O, we give highest priority to filesystem metadata, user tables, log files, and temporary tables; all of these classes are managed as a single class (they share an LRU list). Index files are given lowest priority. Unused indexes can consume a considerable amount of cache space and, in these tests, are served from disk sufficiently fast. We discovered this when we first began analyzing the DB I/O requests in our storage system. That is, the classified I/O both identifies the opportunity for cache optimization, and it provides a means by which the optimization can be realized.

Figure 5 compares the cache contents of LRU and LRU-S. For the LRU test, most of the cache is consumed by index files; user tables and temporary tables consume the remainder. Because index files are created after the DB is created, it is understandable why they consume such a large portion of the cache. In contrast, LRU-S fences off the index files, leaving more cache space for user tables, which are often accessed randomly.

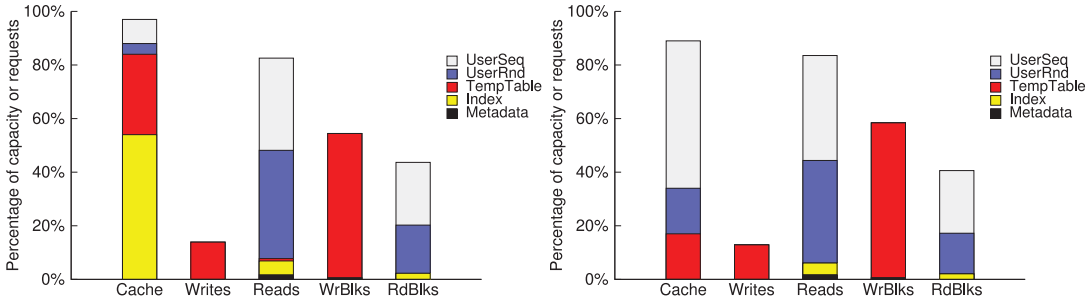
The end result is an improved cache hit rate (Figure 6a), slightly less cache cleaning overhead (Figure 6b), and a 20% improvement in query time (Figure 6c). The non-cached run completes all 4 queries in 680 seconds, LRU in 463 seconds, and LRU-S in 386 seconds. Also, unlike the file and e-mail server runs, we see more variance in TPC-H running time when not using LRU-S. This applies to both the non-cached run and the LRU run. Because of this, we average over three runs and include error bars. As seen in Figure 6c, LRU-S not only runs faster, but it also reduces performance outliers.

6. RELATED WORK

File and storage system QoS is a heavily researched area. Previous work focuses on QoS guarantees for disk I/O [54], QoS guarantees for filesystems [4], configuring storage systems to meet performance goals [55], allocating storage bandwidth to application classes [46], and mapping administrator-specified goals to appropriate storage system designs [48]. In contrast, we approach the QoS problem with I/O classification, which benefits from a coordinated effort between the computer system *and* the storage system.

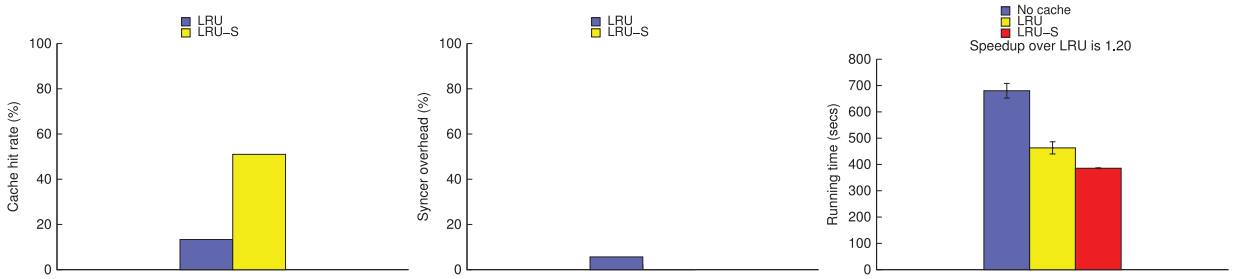
More recently, providing performance differentiation (or isolation) has been an active area of research due to the increasing level in which storage systems are being shared within a data center. Such techniques manage I/O scheduling to achieve fairness within a shared storage system [17, 50, 53]. The work presented in this paper provides a finer granularity of control (classes) for such systems.

Regarding caching, numerous works focus on flash and its integration into storage systems as a conventional cache [20, 23, 24]. However, because enterprise workloads often exhibit such poor locality of reference, it can be difficult to make conventional caches cost-effective [31]. In contrast, we show



(a) LRU cache and I/O breakdown (b) LRU-S cache and I/O breakdown

Figure 5: TPC-H results. Cache contents and breakdown of blocks written/read.



(a) Read hit rate (b) Syncer overhead (c) TPC-H performance

Figure 6: TPC-H performance indicators

that *selective* caching, even when applied to the simplest of caching algorithms (LRU) can be cost effective. Though we introduce selective caching in the context of LRU [39], any of the more advanced caching algorithms could be used, such as LRU-K [32], CLOCK-Pro [13], 2Q [15], ARC [25], LIRS [14], FBR [35], MQ [59], and LRFU [19].

Our block-level selective caching approach is similar to FS-level approaches, such as Conquest [51] and zFS [36], where faster storage pools are reserved for metadata and small files. And there are other block-level caching approaches with similar goals, but different approaches. In particular, Hystor [6] uses data migration to move metadata and other latency sensitive blocks into faster storage, and Karma [57] relies on *a priori* hints on database block access patterns to improve multi-level caching.

The characteristics of flash [7] make it attractive as a medium for persistent transactions [34], or to host flash-based filesystems [16]. Other forms of byte-addressable non-volatile memory introduce additional filesystem opportunities [8].

Data migration [1, 2, 5, 6, 18, 21, 56], in general, is a complement to the work presented in this article. However, migration can be expensive [22], so it is best to allocate storage from the appropriate storage during file creation, whenever possible. Many files have well-known patterns of access, making such allocation possible [26].

And we are not the first to exploit semantic knowledge in the storage system. Most notably, semantically-smart disks [43] and type-safe disks [40, 58] explore how knowledge of on-disk data structures can be used to improve performance, reliability, and security. But we differ, quite fundamentally, in that we send higher-level semantic information with each I/O request, rather than detailed block information (e.g., inode structure) through explicit management commands. Further, unlike this previous work, we do not offload block management to the storage system.

7. CONCLUSION

The inexpressive block interface limits I/O optimization, and it does so in two ways. First, computer systems are having difficulty optimizing around complex storage system internals. RAID, caching, and non-volatile memory are good examples. Second, storage systems, due to a lack of semantic information, experience equal difficulty when trying to optimize I/O requests.

Yet, an entire computer industry has been built around blocks, so major changes to this interface are, today, not practical. Differentiated Storage Services addresses this problem with I/O classification. By adding a small classifier to the block interface, we can associate QoS policies with I/O classes, thereby allowing computer systems and storage system to agree on shared, block-level policies. This will enable continued innovation on both sides of the block interface.

Our filesystem prototypes show significant performance improvements when applied to storage system caching, and our database proof-of-concept suggests similar improvements.

We are extending our work to other realms such as reliability and security. Over time, as applications come to expect differentiated service from their storage systems, additional usage models are likely to evolve.

Acknowledgments

We thank our Intel colleagues who helped contribute to the work presented in this paper, including Terry Yoshii, Mathew Eszenyi, Pat Stolt, Scott Burridge, Thomas Barnes, and Scott Hahn. We also thank Margo Seltzer for her very useful feedback.

8. REFERENCES

[1] M. Abd-El-Malek, W. V. C. II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa Minor: versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, December 2005. The USENIX Association.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18nd ACM Symposium on Operating Systems Principles (SOSP 01)*, Chateau Lake Louise, Banff, Canada, October 2001.

[3] S. N. I. Association. A Dictionary of Storage Networking Terminology. http://www.snia.org/education/dictionary.

[4] P. R. Barham. A Fresh Approach to File System Quality of Service. In *Proceedings of the IEEE 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 97)*, St. Louis, MO, May 1997.

[5] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 09)*, San Francisco, CA, February 2009. The USENIX Association.

[6] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*, Tucson, AZ, May 31 - June 4 2011.

[7] F. Chen, D. A. Koufaty, and X. Zhang. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2009)*, Seattle, WA, June 2009. ACM Press.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*, Big Sky, MT, October 2009.

[9] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007. The USENIX Association.

[10] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems (TODS)*, 9(4):560–595, December 1984.

[11] H. Huang, A. Hung, and K. G. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of 20th ACM Symposium on Operating System Principles*, pages 263–276, Brighton, UK, October 2005. ACM Press.

[12] Intel Corporation. Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[13] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC 2005)*, Anaheim, CA, April 10-15 2005. The USENIX Association.

[14] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2002)*, Marina Del Rey, CA, June 15-19 2002. ACM Press.

[15] T. Johnson and D. Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago Chile, Chile, September 12-15 1994. Morgan Kaufmann.

[16] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. DFS: A File System for Virtualized Flash Storage. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. The USENIX Association.

[17] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: performance differentiation for storage systems using adaptive control. *ACM Transactions on Storage*, 1(4):457–480, November 2006.

[18] S. Khuller, Y.-A. Kim, and Y.-C. J. Wan. Algorithms for data migration with cloning. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, San Diego, CA, June 2003. ACM Press.

[19] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001.

[20] A. Leventhal. Flash storage memory. In *Communications of the ACM*, volume 51(7), pages 47–51, July 2008.

[21] C. Lu, G. A. Alvarez, and J. Wilkes. Aqueduct: online data migration with performance guarantees. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[22] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 10)*, San Jose, CA, February 2010. The USENIX Association.

[23] B. Marsh, F. Douglis, and P. Krishnan. Flash memory file caching for mobile computers. In *Proceedings of the 27th Hawaii Conference on Systems Science*, Wailea, HI, Jan 1994.

[24] J. Matthews, S. Trika, D. Hensgen, R. Coulson, and K. Grimsrud. Intel Turbo Memory: Nonvolatile disk caches in the storage hierarchy of mainstream computer systems. In *ACM Transactions on Storage (TOS)*, volume 4, May 2008.

[25] N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *IEEE Computer Magazine*, 37(4):58–65, April 2004.

[26] M. Mesnier, E. Thereska, G. Ganger, D. Ellard, and M. Seltzer. File classification in self-* storage systems. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC-04)*, New York, NY, May 2004. IEEE Computer Society.

[27] M. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2007)*, San Diego, CA, June 2007. ACM Press.

[28] M. P. Mesnier, G. R. Ganger, and E. Riedel. Object-based Storage. *IEEE Communications*, 44(8):84–90, August 2003.

[29] D. T. Meyer and W. J. Bolosky. A Study of Practical Deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST 11)*, San Jose, CA, Feb 15-17 2011. The USENIX Association.

[30] O. Muller and D. Graf. Swiss Internet Analysis 2002. http://swiss-internet-analysis.org.

[31] D. Narayanan, E. Thereska, A. Donnelly, S. Elnikety, and A. Rowstron. Migrating Server Storage to SSDs: Analysis of Tradeoffs. In *Proceedings of the 4th ACM European Conference on Computer systems (EuroSys '09)*, Nuremberg, Germany, March 31 - April 3 2009. ACM Press.

[32] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM International Conference on Management of Data (SIGMOD '93)*, Washington, D.C., May 26-28 1993. ACM Press.

[33] PostgreSQL Global Development Group. Open source database. http://www.postgresql.org.

[34] V. Prabhakaran, T. L. Rodeheffer, and L. Zhou. Transactional Flash. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. The USENIX Association.

[35] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1990)*, Boulder, CO, May 22-25 1990. ACM Press.

[36] O. Rodeh and A. Teperman. zFS - A Scalable Distributed File System Using Object Disks. In *Proceedings of the 20th Goddard Conference on Mass Storage Systems (MSS'03)*, San Diego, CA, April 2003. IEEE.

[37] C. Ruemmler and J. Wilkes. Disk shuffling. Technical Report HPL-91-156, Hewlett-Packard Laboratories, October 2001.

[38] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST 02)*, Monterey, CA, January 2002. The USENIX Association.

[39] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating Systems Concepts*. Wiley, 8th edition, 2009.

[40] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe Disks. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. The USENIX Association.

[41] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 04)*, pages 379–394, San Francisco, CA, December 2004. The USENIX Association.

[42] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, pages 15–30, San Francisco, CA, March 2004. The USENIX Association.

[43] 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 2th USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March-April 2003. The USENIX Association.

[44] Standard Performance Evaluation Corporation. Spec sfs. http://www.storageperformance.org.

[45] M. Stonebraker. Operating system support for database management. *Communications of the ACM*, 2(7):412–418, July 1981.

[46] V. Sundaram and P. Shenoy. A Practical Learning-based Approach for Dynamic Storage Bandwidth Allocation. In *Proceedings of the Eleventh International Workshop on Quality of Service (IWQoS 2003)*, Berkeley, CA, June 2003. Springer.

[47] Transaction Processing Performance Council. TPC Benchmark H. http://www.tpc.org/tpch.

[48] S. Uttamchandani, K. Voruganti, S. Srinivasan, J. Palmer, and D. Pease. Polus: Growing Storage QoS Management Beyond a "4-Year Old Kid". In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST 04)*, San Francisco, CA, March 2004. The USENIX Association.

[49] M. Uysal, G. A. Alvarez, and A. Merchant. A modular, analytical throughput model for modern disk arrays. In *Proceedings of the 9th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2001)*, Cincinnati, OH, August 2001. IEEE/ACM.

[50] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[51] A.-I. A. Wang, P. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better performance through a Disk/Persistent-RAM hybrid file system. In *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC 2002)*, Monterey, CA, June 2002. The USENIX Association.

[52] M. Wang, K. Au, A. Ailamaki, A. Brockwell, C. Faloutsos, and G. R. Ganger. Storage device performance prediction with CART models. In *Proceedings of the 12th International Symposium on Modeling Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS-2004)*, Volendam, The Netherlands, October 2004. IEEE.

[53] Y. Wang and A. Merchant. Proportional share scheduling for distributed storage systems. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[54] R. Wijayaratne and A. L. N. Reddy. Providing QoS guarantees for disk I/O. *Multimedia Systems*, 8(1):57–68, February 2000.

[55] J. Wilkes. Traveling to Rome: QoS specifications for automated storage system management. In *Proceedings of the 9th International Workshop on Quality of Service (IWQoS 2001)*, Karlsruhe, Germany, June 2001.

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

[57] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel cAche. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.

[58] C. Yalamanchili, K. Vijayasankar, E. Zadok, and G. Sivathanu. DHIS: discriminating hierarchical storage. In *Proceedings of The Israeli Experimental Systems Conference (SYSTOR 09)*, Haifa, Israel, May 2009. ACM Press.

[59] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 25-30 2001. The USENIX Association.