I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance

Ricardo Koller rkoll001@cs.fiu.edu
Raju Rangaswami raju@cs.fiu.edu
School of Computing and Information Sciences, Florida International University

Abstract

Duplication of data in storage systems is becoming increasingly common. We introduce I/O Deduplication, a storage optimization that utilizes content similarity for improving I/O performance by eliminating I/O operations and reducing the mechanical delays during I/O operations. I/O Deduplication consists of three main techniques: *content-based caching*, *dynamic replica retrieval*, and *selective duplication*. Each of these techniques is motivated by our observations with I/O workload traces obtained from actively-used production storage systems, all of which revealed surprisingly high levels of content similarity for both stored and accessed data. Evaluation of a prototype implementation using these workloads revealed an overall improvement in disk I/O performance of 28-47% across these workloads. Further breakdown also showed that each of the three techniques contributed significantly to the overall performance improvement.

1 Introduction

Duplication of data in primary storage systems is quite common due to the technological trends that have been driving storage capacity consolidation. The elimination of duplicate content at both the file and block levels for improving storage space utilization is an active area of research [7, 17, 19, 22, 30, 31, 41]. Indeed, eliminating most duplicate content is inevitable in capacity-sensitive applications such as archival storage for cost-effectiveness. On the other hand, there exist systems with moderate degree of content similarity in their primary storage such as email servers, virtualized servers, and NAS devices running file and version control servers. In case of email servers, mailing lists, circulated attachments and SPAM can lead to duplication. Virtual machines may run similar software and thus create co-located duplicate content across their virtual disks. Finally, file and version control systems servers of collaborative groups often store copies of the same documents, sources and executables. In such systems, if the degree of content similarity is not overwhelming, eliminating duplicate data may not be a primary concern.

Gray and Shenoy have pointed out that given the technology trends for price-capacity and price-performance of memory/disk sizes and disk accesses respectively, disk data must "cool" at the rate of 10X per decade [11]. They suggest data replication as a means to this end. An instantiation of this suggestion is *intrinsic* replication of data created due to consolidation as seen now in many storage systems, including the ones illustrated earlier. Here, we refer to intrinsic (or application/user generated) data replication as opposed to forced (system generated) redundancy such as in a RAID-1 storage system. In such systems, capacity constraints are invariably secondary to I/O performance.

We analyzed on-disk duplication of content and I/O traces obtained from three varied production systems at FIU that included a virtualized host running two department web-servers, the department email server, and a file server for our research group. We made three observations from the analysis of these traces. First, our analysis revealed significant levels of both *disk static similarity* and *workload static similarity* within each of these systems. Disk static similarity is an indicator of the amount of duplicate content in the storage medium, while workload static similarity indicates the degree of on-disk duplicate content accessed by the I/O workload. We define these similarity measures formally in § 2. Second, we discovered a consistent and marked discrepancy between *reuse distances* [23] for sector and content in the I/O accesses on these systems indicating that content is reused more frequently than sectors. Third, there is significant overlap in content accessed over successive intervals of longer time-frames such as days or weeks.

Based on these observations, we explore the premise that intrinsic content similarity in storage systems and access to replicated content within I/O workloads can both be utilized to improve I/O performance. In doing so, we design and evaluate I/O Deduplication, a storage optimization that utilizes content similarity to either eliminate I/O operations altogether or optimize the resulting disk head movement within the storage system. I/O Deduplication comprises three key techniques: (i) *content-based caching* that uses the popularity of "data

Workload type	File System size [GB]	Memory size [GB]	Reads [GB]			Writes [GB]			File System accessed
			Total	Sectors	Content	Total	Sectors	Content	
web-vm	70	2	3.40	1.27	1.09	11.46	0.86	4.85	2.8%
mail	500	16	62.00	29.24	28.82	482.10	4.18	34.02	6.27%
homes	470	8	5.79	2.40	1.99	148.86	4.33	33.68	1.44%

Table 1: **Summary statistics of one week I/O workload traces obtained from three different systems.**

content" rather than "data location" of I/O accesses in making caching decisions, (ii) *dynamic replica retrieval* that upon a cache miss for a read operation, dynamically chooses to retrieve a content replica which minimizes disk head movement, and (iii) *selective duplication* that dynamically replicates frequently accessed content in scratch space that is distributed over the entire storage medium to increase the effectiveness of dynamic replica retrieval.

We evaluated a Linux implementation of the I/O Deduplication techniques for workloads from the three systems described earlier. Performance improvements measured as the reduction in total disk busy time in the range 28-47% were observed across these workloads. We measured the influence of each technique of I/O Deduplication separately and found that each technique contributed substantially to the overall performance improvement Particularly, content-based caching increased memory caching effectiveness by at least 10% and by as much as 4X in cache hit rate for read operations. Head-position aware dynamic replica retrieval directed I/O operations to alternate locations on-the-fly and additionally reduced average I/O times by 10-20%. And finally, selective duplication created additional replicas of popular content during periods of low foreground I/O activity to further improved the effectiveness of dynamic replica retrieval, leading to a reduction in average I/O times by 23-35%. We also measured the memory and CPU overheads of I/O Deduplication and found these to be nominal.

In Section 2, we make the case for I/O deduplication. We elaborate on a specific design and implementation of its three techniques in Section 3. We perform a detailed evaluation of improvements and overhead for three different workloads in Section 4. We discuss related research in Section 5, discuss salient design and deployment alternatives in Section 6, and finally conclude with directions for future work.

2 Motivation and Rationale

In this section, we investigate the nature of content similarity and access to duplicate content using workloads from three production systems that are in active, daily use at the FIU Computer Science department. We collected I/O traces downstream of an active page cache from each system for a duration of three weeks. These systems have different I/O workloads that consist of a virtual machine running two web-servers (*web-vm* workload), an email server (*mail* workload), and a file server (*homes* workload). The *web-vm* workload is collected from a virtualized system that hosts two CS department web-servers, one hosting the department's online course management system and the other hosting the department's web-based email access portal; the local virtual disks which were traced only hosted root partitions containing the OS distribution, while the http data for these web-servers reside on a network-attached storage. The *mail* workload serves user INBOXes for the entire Computer Science department at FIU. Finally, the *homes* workload is that of a NFS server that serves the home directories of our small-sized research group; activities represent those of a typical researcher consisting of software development, testing, and experimentation, the use of graph-plotting software, and technical document preparation.

Key statistics related to these workloads are summarized in Table 1. The mail server is a heavily used system and generates a highly-intensive I/O workload in comparison to the other two. However, some uniform trends can be observed across these workloads. A fairly small percentage of the total file system data is accessed during the entire week (1.44-6.27% across the workloads), representing small working sets. Further, these are write-intensive workloads. While it is therefore important to optimize write I/O operations, we also note that most writes are committed to persistent storage in the background and do not affect user-perceived performance directly. Optimizing read operations, on the other hand, has a direct impact on user-perceived performance and system throughput because this reduces the waiting time for blocked foreground I/O operations. For read I/O's, we observe that in each workload, the unique content accessed is lesser than the unique locations that are accessed on the storage device. These observation directly motivates the three techniques of our approach as we elaborate next.

2.1 Content-based cache

The systems of interest in our work are those in which there are patterns of work shared across more than one mechanism within a single system. A *mechanism* represents any active entity, such as a single thread or process or an entire virtual machine. Such duplicated mech-

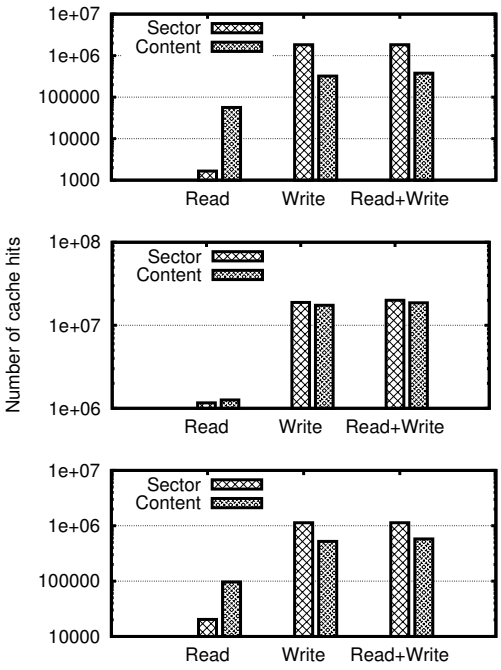

Figure 1: **Page cache hits for the web-vm (top), mail (middle), and homes (bottom) workloads.** *A single day trace was used with an infinite cache assumption.*

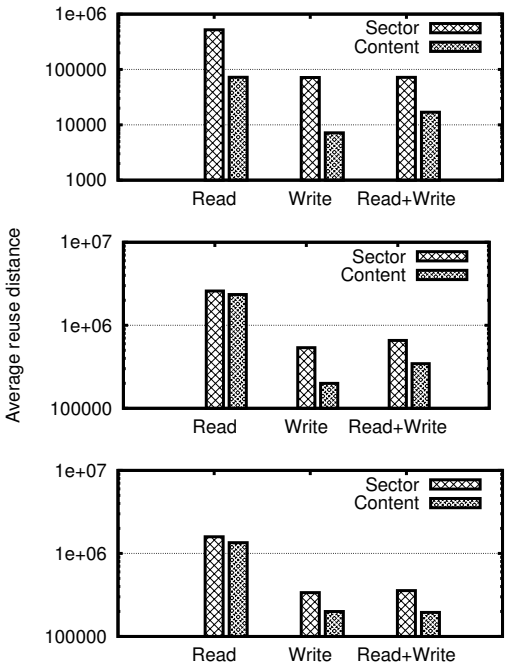

Figure 2: **Contrasting content and sector reuse distances for the web-vm (top), mail (middle), and homes (bottom) workloads.**

anisms also lead to intrinsic duplication in content accessed within the respective mechanisms' I/O operations. Duplicate content, however, may be independently managed by each mechanism and stored in distinct locations on a persistent store. In such systems, traditional storage-location (sector) addressed caching can lead to content duplication in the cache, thus reducing the effectiveness of the cache.

Figure 1 shows that cache hit ratio (for read requests) can be improved substantially by using a content-addressed cache instead of a sector-addressed one. While write I/Os leading to content hits could be eliminated for improved performance, we do not explore it in this paper. A greater number of sector hits with write I/Os are due to journaling writes by the file system, repeatedly overwriting locations within a circular journal space.

For further analysis, we define the *average sector reuse distance* for a workload as the average number of requests between successive requests to the same sector. The *average content reuse distance* is defined similarly over accesses to the same content. Figure 2 shows that the average reuse distance for content is smaller than for sector for each of the three workloads that we studied for both read and write requests. For such workloads, data addressed by content can be cache-resident for lesser time yet be more effective for servicing read requests than if the same cached data is addressed by location. Write requests on the other hand do not depend on cache hits since data is flushed to rather than requested from the storage system. These observations and those from Figure 1 motivate *content-based caching* in I/O Deduplication.

2.2 Dynamic replica retrieval

Systems with intrinsic duplication of mechanism may also operate on duplicate data stored in the persistent stores managed by each mechanism. Such intrinsic content duplication creates opportunities for optimizing I/O operations.

We define the *disk static similarity* as the average number of copies per filesystem-aligned block of content, typically of size 4KB, as a formal measure of content similarity in the storage system. The disk static similarity is calculated as $(all - zeros)/(unique - 1)$, where all is the total number of blocks, $zeroes$ are the number of zeroed blocks (never-used), and $unique$ is the number of blocks with $unique$ content (after eliminating duplicates). This static similarity measure includes blocks that are not currently in use by the file-system; we include such blocks because they were previously used and therefore may contain the same content as in-use data blocks. Table 2 summarizes static similarity values for each of the three workloads. We notice that there is substantial duplication of content on the disks used by each of these workloads. In the case of the *mail* workload, one might expect a higher level of content similarity due to

Workloads	web-vm	mail	homes
Unique pages (millions)	1.9	27	62
Total pages (millions)	5.2	73	183
Static similarity	2.67	2.64	2.94

Table 2: **Disk static similarity.** *Total pages excludes zero pages; Unique pages excludes repeated pages in addition to zero pages.*

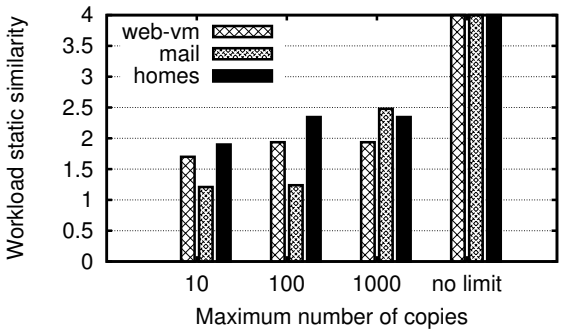

Figure 3: **Workload static similarity.** *One day traces were used. The x axis limits the static similarity consideration to blocks which have at most x copies on disk.*

mailing-list emails and circulated attachments appearing in many INBOXes. However, we point out that all emails within a user's INBOX are managed as a single large file by mail server and therefore individual emails are less likely to be aligned to the filesystem block-size, impacting the disk static similarity measure. Nevertheless, the level of content similarity in these systems is high.

While the presence of substantial duplicate content on each of these systems is promising, it is possible that duplicate content is not accessed frequently in the actual I/O workload. We measured the average number of copies in the storage system for all the blocks read within each of these workloads. We refer to this measure as the *workload static similarity*. By considering only the on-disk duplicate content pertinent to the workload we can better estimate the impact of optimizations based on content similarity. To improve the accuracy our measure, we limit the number of copies of target content. This allows us to prevent a small set of highly replicated content from inflating the workload static similarity value. As shown in Figure 3, the workload static similarity limited to content not repeated more than 1000 times is 2.5. While more than one copy of blocks read is present in the storage system on an average, we note that the disk static similarity values (in Table 2) do overestimate the performance improvement potential.

Based on these observations, we can hypothesize that for each of these workloads, accesses to data that is duplicated on the storage device can be optimally redirected

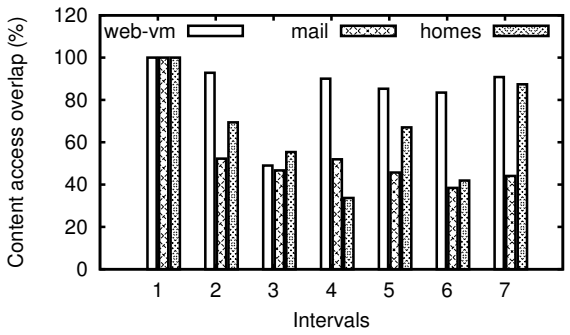

Figure 4: **Content working-sets for three week traces.** *The trace duration is divided into 7 3-day intervals and read content overlap for each interval with all content from the previous interval is presented.*

to the location that minimizes the mechanical overhead of disk I/O operations. This motivates *dynamic replica retrieval* in our approach.

2.3 Selective Duplication

A third property of workloads is repeated access to the same content. Here, we refer to accesses to specific content, which is a different measure than repeated access to the same block address. To illustrate this difference, accesses to two copies of the same executable stored within two virtual disks owned by distinct virtual machines do not lead to repeated access to the same block, but do result in repeated access to the same content.

In Figure 4, we illustrate the overlap in content being accessed across time for each of the workloads using traces over a longer, three week duration. More specifically, we divide the three week trace duration into seven, 3-day intervals and measure the overlap in content read (thus, we exclude writes) within each interval with all data accessed (both read and written) in the previous interval. The first 3-day interval uses self-similarity and therefore represents a 100% content overlap. For the remaining intervals we observe high levels of overlap in the content being read within each interval with all data accessed during the previous interval; average overlaps are 45%, 85%, and 60%, for the mail, web-vm, and homes workloads respectively.

Based on these observation, we can assume that if data accessed in the recent past were replicated in locations dispersed across the disk area, the choice in access provided by such replicas for future I/O operations can help reduce disk arm movement and improve I/O performance. Complementary findings about diurnal patterns in I/O workloads with alternating periods of low and high storage activity [8, 20] suggest that such *selective duplication*, if performed opportunistically during night-time, may result in negligible impact to foreground I/O activity.

3 System Design

I/O Deduplication systematically explores the use of content similarity within storage systems to reduce the mechanical delays incurred in I/O operations and/or to eliminate I/O operations altogether. In this section, we start with an overview of the system architecture and then present the various design choices and rationale behind constructing each of the three mechanisms that constitute I/O Deduplication.

3.1 Architectural Overview

An optimization based on content similarity can be built at various layers of the storage stack, with varying degrees of access and control over storage devices and the I/O workload. Prior research has argued for building storage optimizations in the block layer of the storage stack [12]. We choose the block layer for several reasons. First, the block interface is a generic abstraction that is available in a variety of environments including operating system block device implementations, software RAID drivers, hardware RAID controllers, SAN (e.g., iSCSI) storage devices, and the increasingly popular storage virtualization solutions (e.g., IBM SVC [16], EMC Invista [9], NetApp V-Series [28]). Consequently, optimizations based on the block abstraction can potentially be ported and deployed across these varied platforms. In the rest of the paper, we develop an operating system block device oriented design and implementation of I/O Deduplication. Second, the simple semantics of block layer interface allows easy I/O interception, manipulation, and redirection. Third, by operating at the block layer, the optimization becomes independent of the file system implementation, and can support multiple instances and types of file systems. Fourth, this layer enables simplified control over system devices at the block device abstraction, allowing an elegantly simple implementation of selective duplication that we describe later. Finally, additional I/Os generated by I/O Deduplication can leverage I/O scheduling services, thereby automatically addressing the complexities of block request merging and reordering.

Figure 5 presents the architecture of I/O Deduplication for a block device in relation to the storage stack within an operating system. We augment the storage stack's block layer with additional functionality, which we term *I/O Deduplication layer*, to implement the three major mechanisms: the content-based cache, the dynamic replica retriever, and the selective duplicator. The *content-based cache* is the first mechanism encountered by the I/O workload which filters the I/O stream based on hits in a content-addressed cache. The *dynamic replica retriever* subsequently optionally redirects the unfiltered read I/O requests to alternate locations on the disk to avail the best access latencies to requests. The *selective duplicator* is composed of a *kernel* sub-component that tracks content accesses to create a candidate list of content for replication, and a *user-space* process that runs during periods of low disk activity and populates replica content in scratch space distributed across the entire disk. Thus, while the kernel components run continuously, the user-space component runs sporadically. Separating out the actual replication process into a user-level thread allows greater user/administrator control over the timing and resource consumption of the replication process, an I/O resource-intensive operation. Next, we elaborate on the design of each of the three mechanisms within I/O Deduplication.

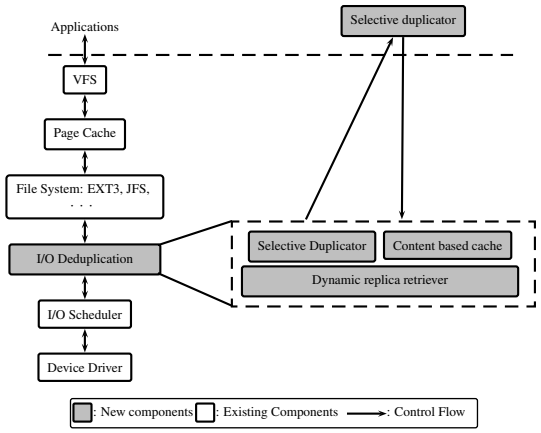

Figure 5: **I/O Deduplication System Architecture.**

3.2 Content based caching

Building a content based cache at the block layer creates an additional buffer cache separate from the virtual file system (VFS) cache. Requests to the VFS cache are sector-based while those to the I/O Deduplication cache are both sector- and content-based. The I/O Deduplication layer only sees the read requests for sector misses in the VFS cache. We discuss exclusivity across these caches shortly. In the I/O Deduplication layer, read requests identified by sector locations are queried against a dual sector- and content-addressed cache for hits before entering the I/O scheduler queue or being merged with an existing request by the I/O scheduler. Population of the content-based cache occurs along both the read and write paths. In case of a cache miss during a read operation, the I/O completion handler for the read request is intercepted and modified to additionally insert the data read into the content-addressed cache after I/O completion only if it is not already present in the cache and is important enough in the LRU list to be cached. A write request to a sector which had contained duplicate data is simply removed from the corresponding duplicate sector list to ensure data consistency for future accesses. The new data contained within write requests is optionally

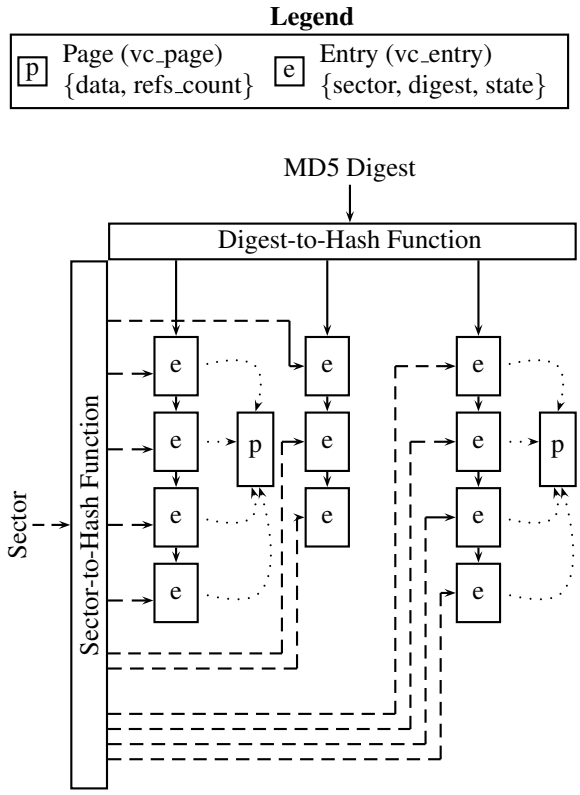

Figure 6: **Data structure for the content-based cache.** *The cache is addressable by both sector and content-hash. vc_entrys are unique per sector. Solid lines between vc_entrys indicates that they may have the same content (they may not in case of hash function collisions.) Dotted lines form a link between a sector (vc_entry) and a given page (vc_page.) Note that some vc_entrys do not point to any page – there is no cached content cached for these. However, this indicates that the linked vc_entrys have the same data on disk. This happens when some of the pages are evicted from the cache. Additionally, pages form an LRU list.*

inserted into the content-addressed cache (if it is sufficiently important) in the onward path before entering the request into the I/O scheduler queue to keep the content cache up-to-date with important data.

The in-memory data structure implementing the content-based cache supports look-up based on both sector and content-hash to address read and write requests respectively. Entries indexed by content-hash values contain a sector-list (list of sectors in which the content is replicated) and the corresponding data if it was entered into the cache and not replaced. Cache replacement only replaces the content field and retains the sector-list in the in-memory content-cache data structure. For read requests, a sector-based lookup is first performed to determine if there is a cache hit. For write requests, a content-hash based look-up is performed to determine a hit and the sector information from the write request is added to the sector-list. Figure 6 describes the data structure used to manage the content-based cache. A write to a sector that is present in a sector-list indexed by content-hash is simply removed from the sector list and inserted into a new list based on the sector's new content hash. It is important to also point out that our design uses a write-through cache to preserve the semantics of the block layer. Next, we discuss some practical considerations for our design.

Since the content cache is a second-level cache placed below the file system page cache or, in case of a virtualized environment, within the virtualization mechanism, typically observed recency patterns in first level caches are lost at this caching layer. An appropriate replacement algorithm for this cache level is therefore one that captures frequency as well. We propose using Adaptive Replacement Cache (ARC) [24] or CLOCK-Pro [18] as good candidates for a second-level content-based cache and evaluate our system with ARC and LRU for contrast.

Another concern is that there can be a substantial amount of duplicated content across the cache levels. There are two ways to address this. Ideally, the content-based cache should be integrated into a higher level cache (e.g., VFS page cache) implementations if possible. However, this might not be feasible in virtualized environments where page caches are managed independently within individual virtual machines. In such cases, techniques that help make in-memory cache content across cache levels exclusive such as cache hints [21], demotions [38], and promotions [10] may be used. An alternate approach is to employ memory deduplication techniques such as those proposed in the VMware ESX server [36], Difference Engine [13], and Satori [25]. In these solutions, duplicate pages within and across virtual machines are made to point to the same machine frame with use of an extra level of indirection such as the shadow page tables. In memory duplicate content across multiple levels of caches is indeed an orthogonal problem and any of the referenced techniques could be used as a solution directly within I/O Deduplication.

3.3 Dynamic replica retrieval

The design of dynamic replica retrieval is based on the rationale that better I/O schedules can be constructed with more options for servicing I/O requests. A storage system with high disk static similarity (i.e., duplicated content) creates such options naturally. With dynamic replica retrieval in such a system, read I/O requests are optionally indirected to alternate locations before entering the I/O scheduler queue. Choosing alternate locations for write requests is complicated due to the need for ensuring up-to-date block content; while we do not con-

sider this possibility further in our work, investigating alternate mechanisms for optimizing write operations to utilize content similarity is certainly a promising area of future work. The content-addressed cache data structure that we explored earlier supports look-up based on sector (contained within a read request) and returns a sector-list that contain replicas of the requested content, thus providing alternate locations to retrieve the data from.

To help decide if and to where a read I/O request should be redirected, the dynamic replica retriever continuously maintains an estimate of the disk head position by monitoring I/O completion events. For estimating head position, we use read I/O completion events only and ignore I/O completion events for write requests since writes may be reported as complete as soon as they are written to the disk cache. Consequently, the head position as computed by the dynamic replica retriever is an approximation, since background write flushes inside the disk are not accounted for. To implement the head-position estimator, the last head position is updated during the execution of the I/O completion handler of each read request. Additionally, the direction of the disk arm managed by the scheduler is also maintained for elevator-based I/O schedulers.

One complication with redirection of an I/O request before a possible merge operation (done by the I/O scheduler later) is that this optimization can reduce the chances for merging the request with another request already awaiting service in the I/O scheduler queue. For each of the workloads we experimented with, we did indeed observe reduction in merging negatively affecting performance when using redirection purely based on current head-position estimates. Request merging should gain priority over any other operation since it eliminates mechanical overhead altogether. One means to prioritize request merging is performing the indirection of requests below the I/O scheduler which performs merging within its mechanisms. Although this is an acceptable and correct solution, it is substantially more complex compared to implementation at the block layer above the I/O scheduler because there are typically multiple dispatch points for I/O scheduler implementations inside the operating system. The second option, and the one used in our system, is to evaluate whether or not to redirect the I/O request to a more opportune location, based on the an actively maintained digest of outstanding requests at the I/O scheduler – these are requests that have been dispatched to the I/O scheduler but not yet reported as completed by the device. If an outstanding request to a location adjacent to the current request exists in the digest, redirection is avoided to allow for merging.

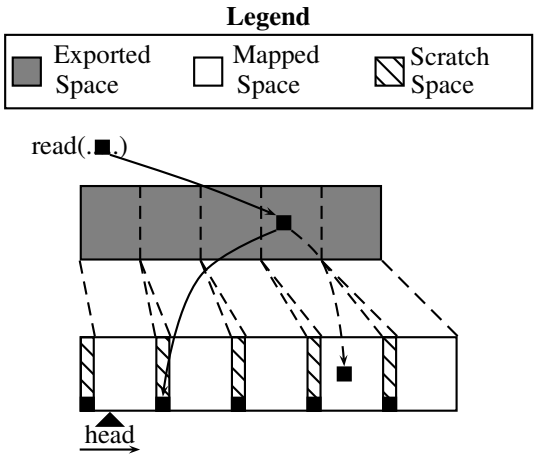

Figure 7: **Transparent replica management for selective duplication.** *The read request to the solid block in the exported space can either be retrieved from its original location in the mapped space or from any of the replicas in the scratch space that reduce head movement.*

3.4 Selective duplication

Figure 4 revealed that the overlap in longer-time frame working sets can be substantial in workloads, more than 80% in some cases. While such overlapping content are the perfect choice for content to be cached, such content was found to be too big to fit in memory.

A complementary optimization to dynamic replica retrieval based on this observation is that an increase in the number of duplicates for popular content on the disk can create even greater opportunities for optimizing the I/O schedule. A basic question then is *what* to duplicate and *when*. We implemented *selective duplication* to run every day during periods of low disk activity based on the observed diurnal patterns in the I/O workloads that we experimented with. The question of what to duplicate can be rephrased as what is the content accessed in the previous days that is likely to be accessed in the future? Our analysis of the workloads revealed that the content overlap between the most frequently used content of the previous days was found to be a good predictor of future accesses to content. The selective duplicator kernel component calculates the list of frequently used content across multiple days by extending the ARC replacement algorithm used for the content-addressed cache.

A list of sectors to duplicate is then forwarded to the user-space replicator process which creates the actual replicas during periods of low activity. The periodic nature of this process ensures that the most relevant content is replicated in the scratch space while older replicas of content that have either been overwritten or are no longer important are discarded. To make the replication process seamless to file system, we implemented *trans-*

parent replica management that implements the scratch space used to store replicas transparently. The scratch space is provisioned by creating additional physical storage volumes/partitions interspersed within the file system data. Figure 7 depicts the transparent replica management wherein the storage is interspersed with five scratch space volumes interspersed between file system mapped space. For file system transparency, a single logically contiguous volume is presented to the file system by the I/O Deduplication extension. The scratch space is used to create one or more replicas of data in the exported space. Since the I/O operations issued during the selective duplication process are themselves routed via the in-kernel I/O Deduplication components, the additional content similarity information due to replication is automatically recorded into the content cache.

3.5 Persistence of metadata

A final issue is the persistence of the in-memory data structure so that the system can retain intelligence about content similarity across system restart operations. Persistence is important for retaining the locations of on-disk intrinsic and artificially created duplicate content so that this information can be restored and used immediately upon a system restart event. We note that while persistence is useful to retain intelligence that is acquired over a period of time, "continuous persistence" of metadata in I/O Deduplication is not necessary to guarantee the reliability of the system, unlike other systems such as the eager writing disk array [40] or doubly distorted mirroring [29]. In this sense, *selective duplication* is similar to the opportunistic replication as performed by FS2 [15] because it tracks updates to replicated data in memory and only guarantees that the primary copy of data blocks are up-to-date at any time. While persistence of the in-memory data is not implemented in our prototype yet, guaranteeing such persistence is relatively straightforward. Before the I/O Deduplication kernel module is unloaded (occuring at the same time the managed file system is unmounted), all in-memory data structure entries can be written to a reserved location of the managed scratch-space. These can then be read back to populate the in-memory metadata upon a system restart operation when the kernel module is loaded into the operating system.

4 Experimental Evaluation

In this section, we evaluate each mechanism in I/O Deduplication separately first and then evaluate their cumulative performance impact. We also evaluate the CPU and memory overhead incurred by an I/O Deduplication system. We used the block level traces for the three systems that were described in detail in § 2 for our evaluation. The traces were replayed as block traces in a similar way as done by blktrace [2]. Blktrace could not be used as-is since it does not record content information; we used a custom Linux kernel module to record content-hashes for each block read/written in addition to other attributes of each I/O request. Additionally, the blktrace tool btreplay was modified to include traces in our format and replay them using provided content. Replay was performed at a maximum acceleration of 100x with care being taken in each case to ensure that block access patterns were not modified as a result of the speedup. Measurements for actual disk I/O times were obtained with per-request block-level I/O tracing using blktrace and the results reported by it. Finally, all trace playback experiments were performed on a single Intel(R) Pentium(R) 4 CPU 2.00GHz machine with 1 GB of memory and a Western Digital disk WD5000AAKB-00YSA0 running Ubuntu Linux 8.04 with kernel 2.6.20.

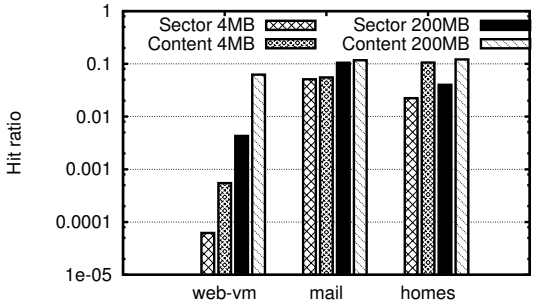

Figure 8: **Per-day page cache hit ratio for content- and sector- addressed caches for read operations.** *The total number of pages read are 0.18, 2.3, and 0.23 million respectively for the web-vm, mail and homes workloads. The numbers in the legend next to each type of addressing represent the cache size.*

4.1 Content based cache

In our first experiment, we evaluated the effectiveness of a content-addressed cache against a sector-addressed one. The primary difference in implementation between the two is that for the sector-addressed cache, the same content for two distinct sectors will be stored twice. We fixed the cache size in both variants to one of two different sizes, 1000 pages (4MB) and 50000 pages (200MB). We replayed two weeks of the traces for each of the three workloads; the first week warmed up the cache and measurements were taken during the second week. Figure 8 shows the average per-day cache hit counts for read I/O operations during the second week when using an *adaptive replacement cache* (ARC) in two modes, content and sector addressed.

This experiment shows that there is a large increase in per-day cache hit counts for the web and the home work-

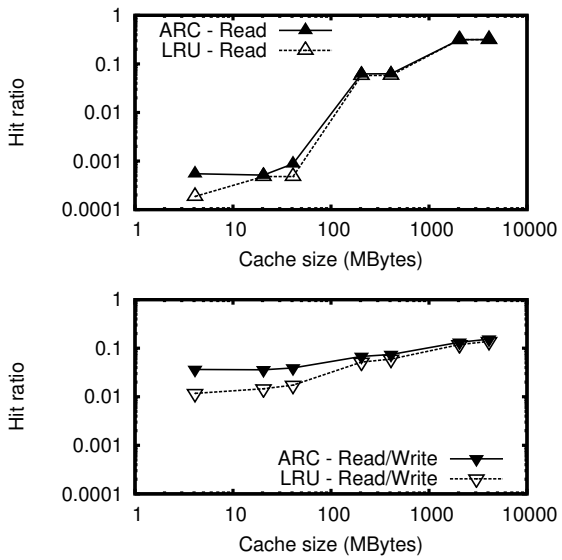

Figure 9: **Comparison of ARC and LRU content based caches for pages read only (top) and pages read/write operations (bottom).** *A single day trace (0.18 million page reads and 2.09 million page read/writes) of the web workload was used as the workload.*

loads when a content-addressed cache is used (relative to a sector-addressed cache). The first observation is that improvement trends are consistent across the two cache sizes. Both caches implementations benefit substantially from a larger cache size except for the *mail* workload, indicating that *mail* is not a cache-friendly workload validated by its substantially larger working set and workload I/O intensity (as observed in Section 2). The *web-vm* workload shows the biggest increase with an almost 10X increase in cache hits with a cache of 200MB compared to the home workload which has an increase of 4X. The *mail* workload has the least improvement of approximately 10%.

We performed additional experiments to compare an LRU implementation with the ARC cache implementation (used in the previous experiments) using a single day trace of the *web-vm* workload. Figure 9 provides a performance comparison of both replacement algorithms when used for a content-addressed cache. For small and large cache sizes, we observe that ARC is either as good or more effective than LRU with ARC's improvement over LRU increasing substantially for write operations at small to moderate cache sizes. More generally, this experiment suggests that the performance improvements for a content-addressed cache are sensitive to the cache replacement mechanism which should be chosen with care.

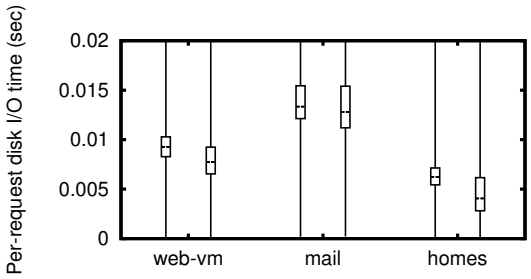

Figure 10: **Improvement in disk read I/O times with dynamic replica retrieval.** *Box and whisker plots depicting median and quartile values of the per-request disk I/O times are shown. For each workload, the values to the left represent the vanilla system and that on the right is with dynamic replica retrieval.*

4.2 Dynamic replica retrieval

To evaluate the effectiveness of dynamic replica retrieval, we replayed a one week trace for each workload with and without using I/O Deduplication. When using I/O Deduplication, prior to replaying the trace workload, information about duplicates was loaded into the kernel module's data structures, as would have been accumulated by I/O Deduplication over the lifetime of all data on the disk. Content-based caching and selective duplication were turned-off. In each case, we measured the per-request disk I/O time per request. A lower per-request disk I/O time informs us of a more efficient storage system.

Figure 10 shows the results of this experiment. For all the workloads there is a decrease in median per-request disk I/O time of at least 10% and up to 20% for the homes workload. These findings indicate that there is room for optimizing I/O operations simply by using pre-existing duplicate content on the storage system.

4.3 Selective duplication

Given the improvements offered by dynamic replica retrieval, we now evaluate the impact of selective duplication, a mechanism whose goal is to further increase the opportunities for dynamic replica retrieval. The workloads and metric used for this experiment were the same as the ones in the previous experiment.

To perform selective duplication, for each workload, ten copies of the predicted popular content were created on scratch space distributed across the entire disk drive. The set of popular data blocks to replicate is determined by the kernel module during the day and exported to user space after a time threshold is reached. A user space program logs the information about the popular content that are candidates for selective duplication and creates the copies on disk based on the information gathered during periods of little or no disk activity. As in the previous

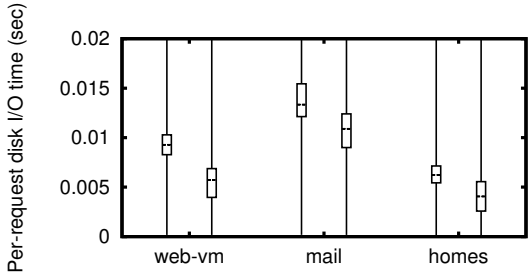

Figure 11: **Improvement in disk read I/O times with selective duplication and dynamic replica retrieval optimizations.** *Other details are the same as Figure 10.*

experiment, prior to replaying the trace workload, all the information about duplicates on disk was loaded into the kernel module's data structures.

Figure 11 (when compared with the numbers in Figure 10) shows how selective duplication improves upon the previous results using pure dynamic replica retrieval. Figure 4 showed that the web workload had more than 80% in content reuse overlap and the effect of duplicating this information can be observed immediately. Overall, the reduction in per-request disk I/O time was improved substantially for the *web-vm* and *homes* workloads, and to a lesser extent for the *homes* workload using this additional technique when compared to using dynamic replica retrieval alone. Overall reductions in median disk I/O times when compared to the vanilla system were 33% for the web workload, 35% for the homes workload, and 23% for mail.

4.4 Putting it all together

We now examine the impact of using all the three mechanisms of I/O Deduplication at once for each workload. We use a sector-addressed cache for the baseline *vanilla* system and a content-addressed one for *I/O Deduplication*. We set the cache size to 200 MB in both cases. Since sector- or content-based caching is the first mechanism encountered by the I/O request stream, the results of the caching mechanism remain unaffected because of the other two, and the cache hit counts remain as with the independent measurements reported in Section 4.1. However, cache hits do modify the request stream presented to the remaining two optimizations. While there is a reduction in the improvements to per-request disk read I/O times with all three mechanisms (not shown) when compared to using the combination of dynamic replica retrieval and selective duplication alone, the total number of I/O requests is different in each case. Thus the average disk I/O time is not a robust metric to measure relative performance improvement. The total disk read I/O time for a given I/O workload, on the other hand, provides an accurate comparative evaluation by taking into account both the reduced number of I/O read operations

Workload	Vanilla (sec)	I/O dedup (sec)	Improvement
web-vm	3098.61	1641.90	47%
mail	4877.49	3467.30	28%
home	1904.63	1160.40	39%

Table 3: **Reduction in total disk read I/O times.**

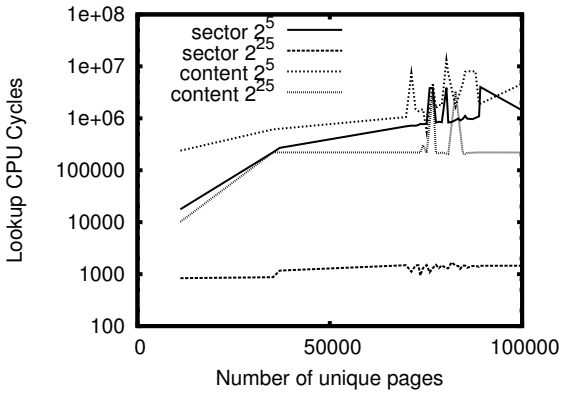

Figure 12: **Overhead of content and sector lookup operations with increasing size of the content-based cache.**

due to content-based caching and the improvements in disk latencies of the latter two optimizations, and effectively measures the true increase in disk I/O efficiency.

When comparing total disk read I/O time for these three workloads, substantial reductions were observed when compared to a vanilla system as shown on Table 3. These uniformly large improvements (28-47% across the three workloads) are a clear indication of the effectiveness of I/O Deduplication in improving I/O performance for a range of different storage workloads.

4.5 Evaluating Overhead

While the gains due to I/O Deduplication are promising, it incurs resource overhead. Specifically, the implementation uses content- and sector- addressed hash-tables to simplify lookup and insert operations into the content based cache. We evaluate the CPU overhead for insert/lookup operations and memory overhead required for managing hash-table metadata in I/O Deduplication.

4.5.1 CPU Overhead

To evaluate the overhead of I/O Deduplication, we measured the average number of CPU cycles required for lookup/insert operations as we vary the number of unique pages (i.e., size) in the content-based cache (i.e., cache size) for a day of the web workload. Figure 13 depicts these overheads for two cache configurations, one configured with 2^{25} buckets in the hash tables and the other with 2^5 buckets. Read operations perform a sector lookup and additionally content lookup in case of a miss

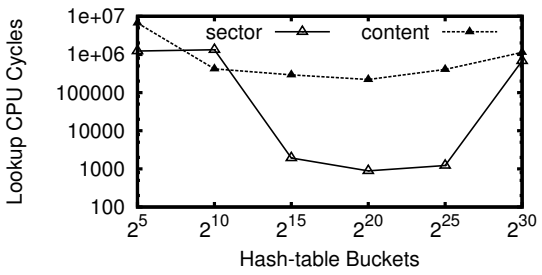

Figure 13: **Overhead of sector and content lookup operations with increasing hash-table bucket entries.**

for insertion. Write operations always perform a sector and content lookup due to our write-through cache design. Content lookups need to first compute the hash for the page contents which takes around 100000 CPU cycles for MD5. With few buckets (2^5) lookup times approach O(N) where N is the size of the hash-table. However, given enough hash-table buckets (2^{25}), lookup times are O(1).

Next, we examined the sensitivity to the hash-table bucket entries. As the number of buckets are increased, the lookup times decrease as expected due to reduction in collisions, but beyond 2^{20} buckets, there is an increase. We attribute this to L2 cache and TLB misses due to memory fragmentation, under-scoring that hash-table bucket sizes should be configured with care. In the sweet spot of bucket entries, the lookup overhead for both sector and content reduces to 1K CPU cycles or less than 1μs for our 2GHz machine. Note that the content lookup operation includes a hash computation which inflates its cycles requirement by at least 100000.

4.6 Memory Overhead

The management of I/O Deduplication's content-based cache introduces memory overhead for managing metadata for the content-based cache. Specifically, the memory overhead is dictated by the size of the cache measured in pages (P), the degree of *Workload static similarity* (WSS), and the configured number of buckets in the hash tables (HTB) which also determine the lookup time as we saw earlier. In our current unoptimized implementation, the memory overhead in bytes (assuming 4 bytes pointers and 4096 bytes pages) :

$$mem(P, WSS, HTB) = 13 * P + 36 * P * WSS + 8 * HTB \quad (1)$$

These overheads include 13 bytes per-page to store the metadata for a a specific page content (*vc_page*), 36 bytes per page per duplicated entry (*vc_entry*), and 8 bytes per hash-table entry for the corresponding linked list. For a 1GB content cache (256K pages), a static similarity of 4, and a hash-table of size 1 million entries, the metadata overhead is ∼48MB or approximately 4.6%.

5 Related Work

In this section, we examine research literature related to workload-based I/O performance optimization and research related to the use of content similarity in memory and storage systems. While there is substantial work done along both these directions, they are for the most part explored as orthogonal techniques in the literature, with the latter primarily being used for optimizing storage capacity utilization using data deduplication.

5.1 I/O performance optimization

Workload-based I/O performance optimization has a long history. The first class of optimizations is based on creating optimized layouts for storage system data. The early works of Wong [37], Vongsathorn *et al.* [35], and Ruemmler and Wilkes [32], which argued for shuffling on-disk data based on data access frequency. Later, Akyurek and Salem [1] argued for copying over shuffling of data with the observation that original layouts are often useful and data popularity and access patterns can be temporary. More recently, ALIS [14] and BORG [3] have employed a dedicated, reorganized area on the disk to improve both locality and sequentiality of I/O access.

The second class of work is based on replicating data and creating opportunities for reducing disk head movement by increasing the number of choices for retrieving data. These include the large body of work on mirroring systems [4]. The work on doubly distorted mirrors [33] creates multiple replicas on master and slave disks to increase both write performance (using initial write-anywhere and background updating of original locations) and read performance by dispatching read requests to the nearest free arm. Zhang *et al.*'s work on eager writing [40] extended this approach to mirrored/striped RAID configurations primarily for database OLTP workload (which are characterized by little locality or sequentiality). Yu *et al.* [39] propose an alternate approach for trading disk capacity for performance in a RAID system, by storing several *rotational replicas* of each block and using a rotational latency sensitive disk scheduler. FS2 [15] proposed replication in file system free-space based on block-access frequency and the use of such selective duplication of content to optimize head movement during subsequent retrieval of replicated data. Quite obviously, selective duplication is motivated by the above works, but is different in two respects: *(i)* it targets identifying replication candidates based on content popularity, rather than block address popularity, and *(ii)* duplication is performed in pre-configured dedicated space transparently to the file system and/or other managers of the storage system. To the best of our knowledge the only work to use content-based optimization of I/O is the work of Tolia *et al.* [34], where the authors use content hashes to perform dynamic replica retrieval choosing be-

tween multiple hosts in an extrinsically-duplicated distributed storage system. Our work, on the other hand, uses intrinsic duplication within a single storage system.

5.2 Data deduplication

Content similarity in both memory and archival storage have been investigated in the literature. Memory deduplication has been explored before in the VMware ESX server [36], Difference Engine [13], and Satori [25], each aiming to eliminate duplicate in-memory content both within and across virtual machines sharing a physical host. Of these, Satori has apparent similarities to our work because it identifies candidates for in-memory deduplication as data is read from storage. Satori runs in two modes: content-based sharing and copy-on-write disk sharing. For content-based sharing, Satori uses content-hashes to track page contents in memory read from disk. Since its goal is not I/O performance optimization, it does not track duplicate sectors on disk and therefore does not eliminate duplicated I/Os that would read the same content from multiple locations. In copy-on-write disk sharing, the disk is already configured to be copy-on-write enabling the sharing of multiple VM disk images on storage. In this mode, duplicated I/Os due to multiple VMs retrieving the same sectors on the shared physical disk would be eliminated in the same way as a regular sector-addressed cache would do. In contrast, our work targets I/O performance optimization by either eliminating I/Os if it were to retrieve duplicate content irrespective of where it may reside on storage or reducing head movement otherwise Thus, the contributions of Satori are complementary to our work and can be used simultaneously.

Data deduplication in archival storage has also gained importance in both the research and industry communities. Current research on data deduplication uses several techniques to optimize the I/O overheads incurred due to data duplication. Venti [30] proposed by Quinlan and Dorward was the first to propose the use of a content-addressed storage for performing data deduplication in an archival system. The authors suggested the use of an in-memory content-addressed index of data to speed up lookups for duplicate content. Similar content-addressed caches were used in data backup solutions such as Peabody [26] and Foundation [31]. Content-based caching in I/O Deduplication is inspired by these works. Recent work by Zhu and his colleagues [41] suggests new approaches to alleviate the disk bottleneck via the use of Bloom filters [5] and by further accounting for locality in the content stream. The Foundation work suggests additional optimizations using batched retrieval and flushing of index entries and a log-based approach to writing data and index entries to utilize temporal locality [31]. The work on sparse indexing [22] suggests improvements to Zhu *et al.*'s general approach by exploiting locality in the chunk index lookup operations to further mitigate the disk I/O bottleneck. I/O Deduplication addresses a orthogonal problem, that of improving I/O performance for foreground I/O workload based on the use of duplicates, rather than their elimination. Nevertheless, the above approaches do suggest interesting techniques to optimize the management of a content-addressed index and cache in main-memory that is complementary to and can be used directly within I/O Deduplication.

6 Discussion

Several aspects of I/O Deduplication from design, implementation, and deployment standpoints warrant further discussion. Some of these also suggest avenues for future work.

Multi-disk deployment. In previous sections, we designed and evaluated a single disk implementation of I/O Deduplication. Multi-disk storage deployments in the form of RAID or more complex NAS appliances are common in enterprise data centers. One might question both the utility and effectiveness of the single disk head movement optimizations central to I/O Deduplication in such systems. We believe that head movement optimizations based on content similarity is viable and can enable complementary optimizations by minimizing the unavoidable mechanical delays in any disk-based storage system. The dynamic replica retrieval and selective duplication sub-techniques require further consideration for multi-disk systems. First, these optimizations must be implemented where information about individual disk head positions is available. Such information is available inside the driver for software RAID, in the RAID controller for hardware RAID, and inside the firmware/OS or internal hardware controllers for NAS appliances. Digest information about the outstanding requests and I/O completion events at each disk can then be utilized as in the single disk design. While the optimal location within each disk for each I/O request can be thus compiled, the complementary issue of load balancing across multiple disks must also be addressed. Apart from the well-known queue depth based techniques for load-balancing, alternate solutions such as simultaneous dispatching to multiple disks combined with just-in-time I/O cancellation can also be envisioned where applicable.

Hash collisions. Our design and implementation of I/O Deduplication makes the assumption that MD5 (128 bits) is collision free. Specifically, this assumption is made when the content-hash entry for a new page being written is registered. A similar assumption, for SHA-1 is made for deduplication in archival storage [30] and low-bandwidth network file transfers [27]. While this as-

sumption may be reasonable in several settings, delivering absolute correctness guarantees requires that this assumption be removed. Systems like Foundation [31] additionally include the provision to perform a byte-wise comparison following a hit in the content cache by reading the target location which potentially contains the duplicate data. This, of course, requires an additional I/O operation. The use of a specific hash function or the method of determining duplicate content is not decisive in our design, and these alternatives can be employed if found necessary within the target deployment scenario.

Variable-sized chunks. Our implementation of I/O Deduplication uses fixed size blocks as the basic data unit for determining content similarity. This choice was motivated by our goal of simplified deployment on a variety of block storage systems. Using variable size chunks as units has been demonstrated to be more effective for similarity detection for mostly similar content and similar content at different offsets within a file [6, 27]. This capability is especially important for archival storage where a single backup file is composed of multiple data files stored at different offsets and possibly with partial modifications. We believe that for online storage systems, this may be of lesser concern, except for very specific applications (e.g., a mail server where entire user INBOXes or folders are managed as a single file). Nevertheless, the use of variable sized chunks for I/O deduplication provides an interesting avenue of future work.

7 Conclusions and Future work

System and storage consolidation trends are driving increased duplication of data within storage systems. Past efforts have been primarily directed towards the elimination of such duplication for improving storage capacity utilization. With I/O Deduplication, we take a contrary view that intrinsic duplication in a class of systems which are not capacity-bound can be effectively utilized to improve I/O performance – the traditional Achilles' heel for storage systems. Three techniques contained within I/O Deduplication work together to either optimize I/O operations or eliminate them altogether. An in-depth evaluation of these mechanisms revealed that together they reduced average disk I/O times by 28-47%, a large improvement all of which can directly impact the overall application-level performance of disk I/O bound systems. The content-based caching mechanism increased memory caching effectiveness by increasing cache hit rates by 10% to 4x for read operations when compared to traditional sector-based caching. Head-position aware dynamic replica retrieval directed I/O operations to alternate locations on-the-fly and additionally reduced I/O times by 10-20%. And, selective duplication created additional replicas of popular content during periods of low foreground I/O activity and further improved the effectiveness of dynamic replica retrieval by 23-35%.

I/O Deduplication opens up several directions for future work. One avenue for future work is to explore content-based optimizations for write I/O operations. A possible future direction is to optionally coalesce or even eliminate altogether write I/O operations for content that are already duplicated elsewhere on the disk, or alternatively direct such writes to alternate locations in the scratch space. While the first option might seem similar to data deduplication at a high-level, we suggest a primary focus on the performance implications of such optimizations rather than capacity improvements. Any optimization for writes affects the read-side optimizations of I/O Deduplication and a careful analysis and evaluation of the trade-off points in this design space is important.

Acknowledgments

We thank the anonymous reviewers and our shepherd Ajay Gulati for excellent feedback which improved this paper substantially. We thank Eric Johnson for his help with production server traces at FIU. This work was supported by the NSF grants CNS-0747038 and IIS-0534530 and by DoE grant DE-FG02-06ER25739.

References

[1] Sedat Akyurek and Kenneth Salem. Adaptive Block Rearrangement. *Computer Systems*, 13(2):89–121, 1995.

[2] Jens Axboe. blktrace user guide, February 2007.

[3] Medha Bhadkamkar, Jorge Guerra, Luis Useche, Sam Burnett, Jason Liptak, Raju Rangaswami, and Vagelis Hristidis. BORG: Block-reORGanization for Self-optimizing Storage Systems. In *Proc. of the USENIX File and Storage Technologies*, February 2009.

[4] Dina Bitton and Jim Gray. Disk Shadowing. In *Proc. of the International Conference on Very Large Data Bases*, 1988.

[5] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[6] Sergey Brin, James Davis, and Hector Garcia-Molina. Copy Detection Mechanisms for Digital Documents. In *Proc. of ACM SIGMOD*, May 1995.

[7] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proc. of the USENIX Annual Technical Conference*, June 2009.

[8] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proc. of the USENIX Conference on File and Storage Technologies*, March 2003.

[9] EMC Corporation. EMC Invista. http://www.emc.com/products/software/invista/invista.jsp.

[10] Binny S. Gill. On multi-level exclusive caching: offline optimality and why promotions are better than demotions.

In *Proc. of the USENIX File and Storage Technologies*, Feburary 2008.

[11] Jim Gray and Prashant Shenoy. Rules of Thumb in Data Engineering. *Proc. of the IEEE International Conference on Data Engineering*, February 2000.

[12] Jorge Guerra, Luis Useche, Medha Bhadkamkar, Ricardo Koller, and Raju Rangaswami. The Case for Active Block Layer Extensions. *ACM Operating Systems Review*, 42(6), October 2008.

[13] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. *Proc. of the USENIX OSDI*, December 2008.

[14] Windsor W. Hsu, Alan Jay Smith, and Honesty C. Young. The Automatic Improvement of Locality in Storage Systems. *ACM Transactions on Computer Systems*, 23(4):424–473, Nov 2005.

[15] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic Data Replication In Free Disk Space For Improving Disk Performance And Energy Consumption. In *Proc. of the ACM SOSP*, October 2005.

[16] IBM Corporation. IBM System Storage SAN Volume Controller. http://www-03.ibm.com/systems/storage/software/virtualization/svc/.

[17] N. Jain, M. Dahlin, and R. Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In *Proc. of the USENIX Conference on File And Storage Systems*, 2005.

[18] Song Jiang, Feng Chen, and Xiaodong Zhang. Clock-pro: An effective improvement of the clock replacement. In *Proc. of the USENIX Annual Technical Conference*, April 2005.

[19] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey. Redundancy Elimination Within Large Collections of Files. *Proc. of the USENIX Annual Technical Conference*, 2004.

[20] Andrew Leung, Shankar Pasupathy, Garth Goodson, and Ethan Miller. Measurement and Analysis of Large-Scale Network File System Workloads. *Proc. of the USENIX Annual Technical Conference*, June 2008.

[21] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. Second-tier cache management using write hints. In *Proc. of the USENIX File and Storage Technologies*, 2005.

[22] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. of the USENIX File and Storage Technologies*, February 2009.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[24] Nimrod Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proc. of USENIX File and Storage Technologies*, 2003.

[25] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened Page Sharing. In *Proc. of the Usenix Annual Technical Conference*, June 2009.

[26] Charles B. Morrey III and Dirk Grunwald. Peabody: The Time Travelling Disk. In *Proc. of the IEEE/NASA MSST*, 2003.

[27] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proc. of the ACM SOSP*, October 2001.

[28] Network Appliance, Inc. NetApp V-Series of Heterogeneous Storage Environments. http://media.netapp.com/documents/v-series.pdf.

[29] Cyril U. Orji and Jon A. Solworth. Doubly distorted mirrors. In *Proceedings of the ACM SIGMOD*, 1993.

[30] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Storage. *Proc. of the USENIX File And Storage Technologies*, January 2002.

[31] Sean Rhea, Russ Cox, and Alex Pesterev. Fast, Inexpensive Content-Addressed Storage in Foundation. *Proc. of USENIX Annual Technical Conference*, June 2008.

[32] C. Ruemmler and J. Wilkes. Disk Shuffling. *Technical Report HPL-CSP-91-30, Hewlett-Packard Laboratories*, October 1991.

[33] Jon A. Solworth and Cyril U. Orji. Distorted Mirrors. *Proc. of PDIS*, 1991.

[34] Niraj Tolia, Michael Kozuch, Mahadev Satyanarayanan, Brad Karp, and Thomas Bressoud. Opportunistic use of content addressable storage for distributed file systems. *Proc. of the USENIX Annual Technical Conference*, 2003.

[35] Paul Vongsathorn and Scott D. Carson. A System for Adaptive Disk Rearrangement. *Softw. Pract. Exper.*, 20(3):225–242, 1990.

[36] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *Proc. of USENIX OSDI*, 2002.

[37] C. K. Wong. Minimizing Expected Head Movement in One-Dimensional and Two-Dimensional Mass Storage Systems. *ACM Computing Surveys*, 12(2):167–178, 1980.

[38] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proc. of the USENIX Annual Technical Conference*, 2002.

[39] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. *Proc. of USENIX OSDI*, 2000.

[40] C. Zhang, X. Yu, A. Krishnamurthy, and R. Y. Wang. Configuring and Scheduling an Eager-Writing Disk Array for a Transaction Processing Workload. In *Proc. of USENIX File and Storage Technologies*, January 2002.

[41] Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. *Proc. of the USENIX File And Storage Technologies*, February 2008.