Cooperative Storage-Level De-Duplication for I/O Reduction in Virtualized Data Centers

Min Li[†], Shravan Gaonkar[‡], Ali R. Butt[†], Deepak Kenchammana[†‡], Kaladhar Voruganti[†‡]
[†]Virginia Tech, [‡]Atlantis Computing, [†‡]NetApp,
Email: {limin,butta}@cs.vt.edu, gaonkar@ieee.org, {Deepak.Kenchammana,kaladhar.Voruganti}@netapp.com

Abstract—**Data centers are increasingly being re-designed for workload consolidation in order to reap the benefits of better resource utilization, power savings, and physical space savings. Among the forces driving savings are server and storage virtualization technologies. As more consolidated workloads are concentrated on physical machines — e.g., the virtual density is already very high in virtual desktop environments, and will be driven to unprecedented levels with the fast growing high-core counts of physical servers — the shared storage layer must respond with virtualization innovations of its own such as de-duplication and thin provisioning. A key insight of this paper is that there is a greater synergy between the two layers of storage and server virtualization to exploit block sharing information than was previously thought possible. We reveal this via developing a systematic framework to explore the storage and virtualization servers interactions. We also quantitatively evaluate the I/O bandwidth and latency reduction that is possible between virtual machine hosts and storage servers using real-world trace driven simulation. Moreover, we present a proof of concept NFS implementation that incorporates our techniques to quantify their I/O latency benefits.**

I. Introduction

A major transformation is underway in data centers, where workload consolidation is gaining traction for increasing resource and administrative utilization, and reducing energy consumption, physical space and acquisition costs.

Consolidation is being adopted at every resource level of data centers: (a) under-utilized physical hosts are being replaced with virtualized client technology running on dense core hosts; (b) disparate and siloed storage servers are being replaced with unified storage and storage virtualization to provide shared storage infrastructure over scale-up or scale-out designed systems; (c) IP and storage networks are being converged with the use of unified fabric and protocols such as Fiber Channel over Ethernet (FCoE); (d) disaster recovery (DR) and backup consolidation is being done within and across entire data centers with the rise in the use of fault-tolerant Virtual Machines (VM) and vaulting infrastructures [9]; and (e) administrative tasks are being consolidated through platform standardization (both in hardware, e.g., x86, and in hypervisor layers, e.g., ESX/Hyper-V/Xen/KVM etc.) and the wide-spread adoption of the cloud and outsourcing models. In this paper, we are concerned with the first two aspects of consolidation, namely, the interactions between host and storage server technologies.

Many applications are exploiting this consolidation trend by running multiple applications in VMs that are running on the same physical machine (host). For example, Virtual Desktop Environments (VDE), where a user's desktop is run in a VM (client), creates the opportunity to share common software and underlying hardware resources. However, consolidation of the environment, as a result of the same applications doing similar tasks at the same time, is resulting in extreme workload demand on the associated storage server. Some of these workloads bursts are very predictable, such as boot storms at 9 am, patch storms on first Tuesday of the month, or virus scan storms at 3 am. Data center administrators are handling these problems by over-provisioning resources for handling peak loads [22]. In order to reduce costs, storage solution designers have observed that there is a lot of duplicate data in these environments [26], [15]: 70% in VDE, 35% in file services environment, 30% in SharePoint, 30% in Email Archival, 25% in Document Archival, 25% in Source code Archival and 10% in Audio-Video files. Thus, data de-duplication techniques for VMs on the storage servers for persistently storing data, and on-wire de-duplication boxes [30] to avoid sending duplicate data on the wire is strongly advocated.

The key premise of this paper is that by examining (1) host side caching, (2) data transfer within hosts and between storage server and hosts, and (3) storage server side de-duplication in a holistic manner, we can realize a software-only solution that obviates the need to provision for peak loads without employing extra memory or adding on-wire de-duplication boxes. To achieve this, we design SeaCache [1] in the context of the NFS protocol for client/storage interactions in consolidated data centers. As we discuss in the related work (Section II), most of the prior research has explored optimizations in the above three areas in isolation only, thus SeaCache offers a novel holistic approach to support consolidated workloads.

Specifically, this paper makes the following contributions. We present read/write content sharing algorithms and a collective cache in the context of an integrated framework, SeaCache. Our investigation consists of both a real implementation and simulations. We compare the alternate algorithms using real-world traces: the CIFS trace is a four month trace of the office workloads for a storage controller hosting the work space of 1500 employees; Virtual Desktop Infrastructure

[1]The protocol is so named to express its goal to coordinate the sea of caches deployed in current data centers.

TABLE I
CLASSIFICATION OF RELATED RESEARCH.

Optimization Focus	Strategy	Research Projects
Storage Server	De-duplication	Venti [24], REBL [16], IBM N Series SS [23], De-dup FS [31]
	Optimized Index Structures	De-dup FS [31], Sparse-Indexing [18], Foundation [25]
	Back Reference Tracking	Backlog [19]
Host / Storage Server Interactions	Caching Hints	Exclusive caching [29], Write-hints [17], X-RAY [4]
	Hash-value Passing	CASPER [27], DeDe [5], Pastiche [6]
	Optimize On-wire Transfers	LBFS [21], TAPER [12], Capo [26]
Host	Memory De-duplication, Compression	Difference Engine [8], ESX Server [28], Satori [20], I/O Dedup [15]
	Cooperative Caching	Cooperative Cache [7], LAC [13], Shark [3]

(VDI) trace is a 14 day trace for nine VMs hosting virtual desktop environments; and Test-Dev trace is a trace of a testing and development environment. We also observed the following results during our experimentation:

- Unlike current deployments, where virtualized environments are provisioned for peak loads in order to deal with boot storms (e.g. VDI environments) by the customers, SeaCache allows provisioning for average loads.
- Many solution providers expect their customers to increase the size of the caches either at the hosts or at the storage server in order to deal with peak workloads. SeaCache allows customers to do away with these cache extensions, thus providing for higher system efficiency.
- SeaCache algorithms are more efficient than simple on-wire data transfer solutions, where de-duplication boxes are placed at both source and destination ends to de-duplicate data being transferred across the wire. SeaCache improves the I/O savings by up to 14% and reduces latency by up to 25%.

The rest of the paper is organized as follows. Section II presents the related works. Section III discusses the system architecture and design details. Section IV and V describe our experimental methodology and the results using both simulation and a prototype implementation. Finally, we conclude in Section VI.

II. Related Work

In this paper, we consider the I/O path between the hosts and storage server with the goal to optimize the data read/write operations between them. We classify several related prior research works (Table I) in optimizing this I/O path based on where the optimizations are made, namely: 1) at the host; 2) at the storage server; or 3) during the interactions between the host and the storage server.

Storage Systems De-duplication Management: A number of works have explored identifying and removing redundancy from stored data to optimize storage usage. Venti [24] utilizes SHA hashes on fixed-sized blocks of data to avoid having to store multiple copies of duplicate data for archival storage. REBL [16] introduces the idea of super-fingerprints to further optimize the amount of data needed for identifying duplicates, thus improving performance. The IBM N Series Storage Systems [23] offers a near-line version of de-duplication techniques in a real system implementation. Similarly, De-duplication File System [31] utilizes techniques such as compact in-memory data structures for identifying duplicates, and improved on-disk locality to yield high efficiency.

Research on the storage server side optimizations also include design of advanced data structures to improve de-duplication efficiency, e.g., in large-scale file systems [31], stream processing [18], and user storage [25], etc. Finally, Backlog [19] offers means for efficiently supporting features such as defragmentation and migration in the presence of de-duplication.

These techniques are complementary to SeaCache. We focus on communicating the data de-duplication information between the storage server and the hosts. SeaCache optimizes the data transfer and latency by leveraging this de-duplication information without any dependency on the underlying topology (primary or backup).

Host / Storage Server Interaction Optimizations: The prior art on obtaining better cache utilization at hosts and storage servers by treating them as an integrated unit offers optimizations, such as exclusive caching [29], sharing write-hints [17], and inferring accesses, e.g., with X-RAY [4], which are complementary to our work. The integrated caching frameworks focus on reducing duplicate data between different tiers of storage caches. However, in SeaCache, the main focus is on avoiding transfer of duplicate data between storage servers and client VMs on hosts.

Distributed hash management techniques such as those employed in CASPER [27] and Pastiche [6] deal with storing content hashes at the hosts so that they can choose the most cost effective replica from amongst a set of storage servers. Similarly, de-centralized de-duplication (DeDe) [5] uses techniques where a set of hosts communicate with each other to de-duplicate data. Such techniques can be leveraged in SeaCache both for transferring de-duplication information between the host and storage server, as well as for host buffer management.

Finally, the on-wire bandwidth reduction techniques, e.g., in LBFS [21] and TAPER [12], are different from SeaCache in that these techniques keep track of the data being sent between the hosts and the storage servers and try to not send duplicates. These techniques are not integrated with the host buffer management techniques, and so the hosts can continue to send data requests to the storage servers even if they have the data cached, and similarly the storage server will do the necessary processing associated with a data request even if it has previously sent the data to the host. Capo [26] leverages the fact that most of the VM disk images are the linked clones from a small set of "golden images" and uses a bit-map to eliminate the duplicate read requests. However, unlike SeaCache, Capo cannot detect duplicate reads outside of golden images or duplicate writes.

Host Cache Management: The host side client caching research primarily consists of how to compress and de-duplicate the client cache (e.g., Difference Engine [8], ESX Server [28], and Satori [20]), and how to avoid sending duplicate read data request to disks (e.g., I/O De-duplication [15]). The cache replacement algorithms and the transferring of cache state to the storage server techniques employed in SeaCache are complementary to the previous host side cache de-duplication/compression strategies and can be employed in that context. Moreover, in SeaCache, the de-duplication algorithms on host and storage server can share de-duplication information to eliminate the CPU intensive recomputation of content sharing attributes.

Cooperative caching techniques focus on designing efficient eviction algorithms and meta-data indices to aggregate distributed client caches as a unified cache and to facilitate fast lookup. N-chance forwarding [7] assigns more weight to singlets that have only one copy of data in the cache by forwarding singlets to random peers. LAC [13] forwards the evicted data block to peers based on data reuse distance and dynamic client synchronization. Shark [3] designs a locality aware distributed index to enable clients to locate nearby copies of data. These techniques are orthogonal to SeaCache, as SeaCache introduces the de-duplication concept into the cooperative cache and focuses on how cooperative caching can help to reduce I/O bandwidth consumption.

III. System Design

This section describes the key design aspects of SeaCache and how de-duplication information is cooperatively shared between storage server and hosts.

A. Design Rationale

A traditional approach to I/O bandwidth saving is to not modify the host and storage server software stack, instead to introduce dedicated nodes for de-duplication, i.e., de-dup boxes [21], both at the host and the storage server. These boxes keep track of the data blocks through a content sharing information/index (CSI) database of all the blocks they have been sent and received, and work together to avoid writing multiple copies of data to the disk. A CSI entry usually consists of a block identifier and the corresponding hash value calculated using collision resistant hash functions such as SHA-1 [1]. Similarly, SeaCache assumes that hash collision from SHA-1 is lower than memory bit flip errors due to cosmic rays for all practical purposes.

In the de-dup box approach, the box is a separate entity and it is not aware of the host-side or storage server-side cache contents. Thus, read requests from different clients will

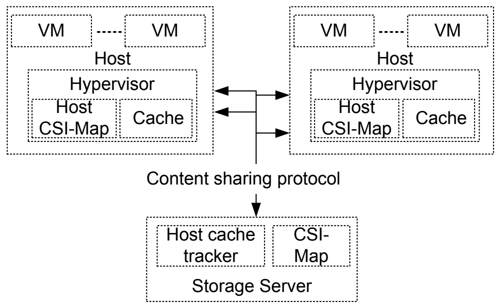

Fig. 1. SeaCache system architecture overview.

always be sent to the storage server even if the data already exists in the host cache. Furthermore, the storage server side block de-duplication information is not leveraged, and thus, this data is maintained separately at the de-dup boxes and at the storage server. By integrating host-side cache with server-side de-duplication, we can explore the opportunity to build a cooperative I/O de-duplication solution between host and storage server.

B. Architecture

Figure 1 shows the overall architecture of SeaCache. The target environment comprises host physical machines with multiple clients (VMs), which interact with a storage server for persistent data storage. The main software components include a specialized *page cache manager* on the host, a *de-duplication system* on the storage server, a storage server *cache-tracker* that keeps track of the host cache contents, and a *protocol for sharing CSI* between the hosts (cooperative caching) and the shared storage server.

When data is written to the storage server, it is de-duplicated (either in-line or as a background process) as follows. The contents are hashed at the granularity of a block, and the hash information is saved in the CSI data structure. The CSI is then compared to and, if not already present, stored in a CSI database. Each entry of the CSI database is a tuple consisting of the logical block number (LBN), and the block's hash value. If a logical block's CSI matches one already in the CSI database, it indicates that the logical block is a duplicate and its contents are not written to the disk. Thus, a physical block could potentially map to multiple logical blocks. The information about mapping of logical blocks to a physical block is maintained by the storage server in a mapping structure, *CSI-Map*, which has an entry for every physical block (in use).

C. Read Protocols in SeaCache

In the following discussion, we present our read protocols using an example physical block usage scenario illustrated in Figure 3.

Basic Protocol When a logical block, e.g., 101, for VM_1 is not found in the host's cache, the host first requests the CSI for the LBN (`Read#`) from the storage server, instead of sending a regular read request. The storage server looks up the associated CSI-Map entry and returns the content identifier, which in

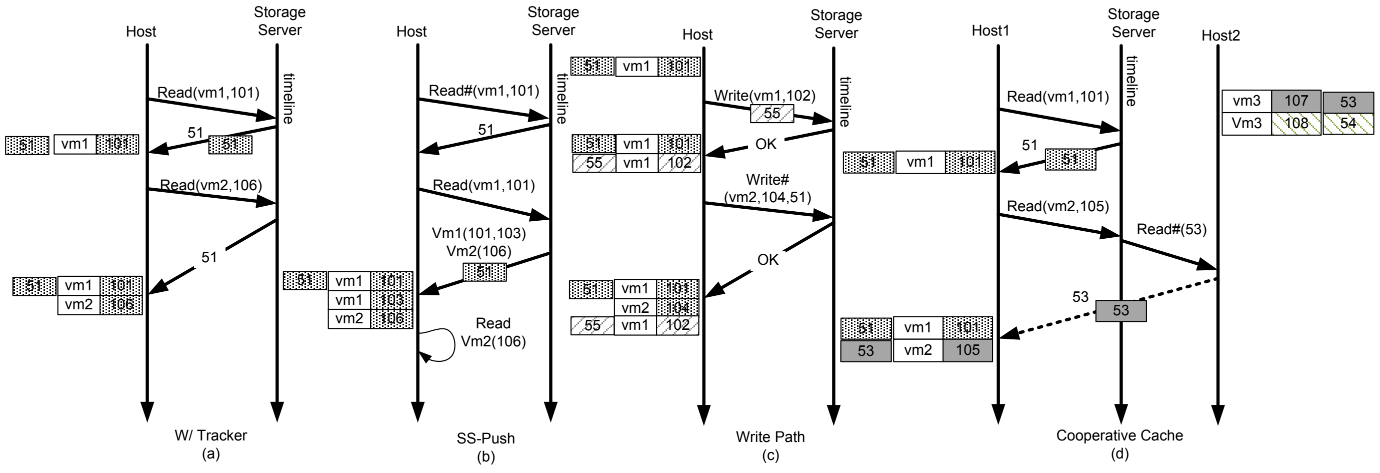

Fig. 2. SeaCache protocols using CSI-Map information.

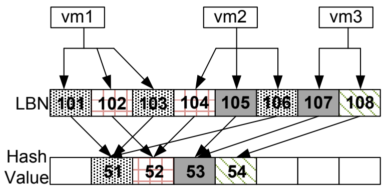

Fig. 3. An example logical to physical block mapping.

this example is Hash 51. The host maintains a local CSI-Map cache, which it uses to determine that Hash 51 is not present at the host. The host then sends an actual data request for LBN 101. Upon receipt of the block from the storage server, the host also updates its local CSI-Map cache to store the mapping information. Later, when the host sends a CSI request for LBN 106 for VM_2, the response Hash 51 is already in the cache, and a subsequent read request to storage server for LBN 106 is avoided.

This basic protocol saves network bandwidth between the host and the storage server by avoiding the necessity to put the data block on the wire. However, it introduces an extraneous round of CSI request messages to be exchanged for every read I/O request missed from client caches.

Protocol with Tracker We can eliminate CSI requests by keeping track of the content of the host cache using the cache-tracker at the storage server. Figure 2(a) displays the storage server-Host protocol using this approach. Here, the host uses a traditional request for reading LBN 101 of VM_1. The storage server replies with data and its hash, Hash 51, back to the host. The storage server learns that Hash 51 is stored in host cache. Later, when the host requests a read for LBN 106 for VM_2, the storage server simply returns the CSI-Map entry that points to the already present Hash 51 in the host cache.

In this approach, we require the storage server to accurately track the host cache contents by the read requests and eviction information. Therefore, whenever the host evicts data from its cache, it needs to inform the storage server by piggybacking eviction information on its next message to the storage server (not shown in the figure). Note that no additional separate message is needed. This approach facilitates the cooperative cache between hosts (detailed in Section III-E). This optimization avoids additional round trip time (RTTs) for CSI requests for which the host has to communicate with the storage server to get the hash value of the requested block.

Maintaining a CSI-Map within the storage server might seem to consume additional memory. However, such tracking is worthwhile because it allows the storage server to leverage the hosts cache space to exploit more de-duplication and thus reduce read I/O. In our design, each data block is 4 kilobytes, while hash-entry is 24 bytes. Therefore, we can map $170\times$ more blocks in cache using the CSI-Map instead of caching the actual data. Handling the additional eviction information for remote cache-tracker of hosts may require additional storage server CPU cycles. However, we argue that there is a large disparity in computational power versus I/O latency. For instance, a 3 GHz processor has 3 million compute cycles to spare for every 1 millisecond of latency from the I/O subsystem. Therefore, we argue that compute overhead is not an issue in this case.

Storage Server Push Another approach to eliminate RTT is to prefetch relevant CSI-Map information from the storage server to the host. This can be achieved either by returning additional CSI-Map information to a host in response to a CSI request, *storage server-Push*, or via an asynchronous callback that sends the associated CSI-Map entry from the storage server to the host. Figure 2(b) illustrates the storage server-Push. In this case, when the host requests the data for LBN 101 for VM_1, the storage server replies with not only the data but also the CSI-Map entry for the associated hash value, which in this case indicates that LBNs 101 and 103 of VM_1 and LBN 106 of VM_2 all map to Hash 51. The extra information is stored in the host's CSI-Map cache. Later, when the host wants to read LBN 106 for VM_2, it already knows that the associated hash value Hash 51 and that its contents are already in the host cache. Thus, no extra CSI request is sent. The CSI-Maps piggybacked by the storage server determines the quality of I/O reduction using this approach. If the storage server is aware of the topology information of the data-center/cloud, the storage server can choose to send back the CSI entry of VMs on a particular host, improving the I/O reduction. Furthermore, it is better to send back the CSI information that resides in the cache only. Thus, this approach is sensitive to the knowledge and understanding of the workload characteristics to obtain optimal performance.

D. Write Protocol in SeaCache

Consistency In virtual environments, there are no shared writes. Each virtual machine will only be able to access the virtual image file attached to it. Therefore, we focus our discussion to such share-nothing environment.

Figure 2(c) demonstrates a basic protocol for I/O reduction on the write path, which is similar to the read path protocol. Initially the VM_1 on the host writes a block with LBN 102 whose original hash value is Hash 52. The host calculates the new hash value Hash 55, determines that it is not in its local cache and sends a request to write the data to the storage server. Now, the second write request by VM_2, LBN 104, with hash value changed from Hash 52 to Hash 51 is a cache hit. In this case, the host sends only the metadata to the storage. If the storage server is able to map the hash value to the actual data, it replies with success and no further action needs to be done. If the storage server replies with failure, the host will now need to send the actual data.

In order to check whether a block represented by the hash value sent back by any host is actually present, the storage server needs to perform a CSI-Map lookup. Most storage servers cannot keep the entire CSI-Map of their blocks in the primary caches. Any design or solution to seek CSI-Map from the secondary-level cache will add additional latency to the I/O request. If the storage server performs I/O reduction on only CSI maps in the primary cache, some write path I/O reduction opportunities will be missed.

By integrating content sharing information into storage server and host, the de-duplication workload can be distributed across hypervisor and storage server. Once the hash value of the blocks are calculated the storage server can simply leverage that information to reduce the usage of computational and disk I/O resources, which in turn can benefit the foreground read/write requests service.

E. Protocol for Cooperative Cache in SeaCache

It is straight forward to expand the read/write CSI protocol to build a cooperative cache between multiple hosts and storage server based on our deployment architecture as described in Section III-B. Cooperative cache aggregates the cache space from all hosts to further distribute the I/O load away from the storage server. To implement such cooperative cache, efficient meta-data lookup and request forwarding mechanisms are needed. The host cache tracker offers an ideal data structure for meta-data lookup. If more than one host is keeping the required data, the storage server can randomly choose one to forward the requests to.

Figure 2 (d) describes the working of SeaCache. When the first read request from host 1 arrives at the storage server, the storage server checks for the block(Hash 51) in its cache. If the block is not found, the server checks its remote cache-tracker for availability of the block in any of the remote hosts. In the above figure, the hash value Hash 51 is not found in the cooperative cache. Therefore, the storage server fetches the data block from disk and sends back both Hash 51 and the data block. When a second request arrives, the storage server determines that the hash value of the requested block is Hash 53, which is not stored in its cache but stored in the cache of host 2. The server delegates the block request to the client at host 2, which then sends the data directly to host 1.

SeaCache requires host 1 to directly receive data from host 2, which is not supported by traditional RPC calls. However, RPC delegation as proposed by Anderson et. al. [2] should suffice as an elegant alternative here. The delegation protocol creates a reply token, allowing the reply token to be relayed from node to node until some node answers the request. Without this technique, we would need two RPC calls one to communicate with the storage sever and another one to communicate with host 2 to get the data.

We can further mitigate the cache-misses in SeaCache protocol by **enhancing the cache replacement algorithm** inside the host. The enhanced LRU or ELRU host cache manager uses CSI information from storage server to determine the block to evict from its cache. Instead of evicting the least recently accessed block, this algorithm also weighs in the sharing count of the block before evicting it. Specifically, we choose the least shared block within the last n blocks to be evicted. A larger n may yield a higher hit ratio but may have a higher eviction overhead. In Section V, we explore the impact of n on the efficiency of the host cache algorithm. This sharing count metric becomes important in context of SeaCache as it is more likely to improve the cache hit for highly referenced blocks.

IV. Experimentation Methodology

In this section, we discuss both a simulator and a proof-of-concept implementation of SeaCache, and the workloads we have used for experimentation.

A. Simulator Framework

In order to test our protocols in a controlled setting and explore the large configuration space, we have built a realistic system-level simulator. Figure 4 shows the modules of the simulator and their interactions.

Trace File Parser: This main module takes preprocessed trace files as input, parses them, and reconstructs the read and write commands to drive the simulation.

Virtual Machine: This main module implements a model of the client VMs. For our tests, in essence, it creates an instance of a traditional LRU cache.

Physical Machine: This is another main module that models a host using specified configuration settings. It supports a

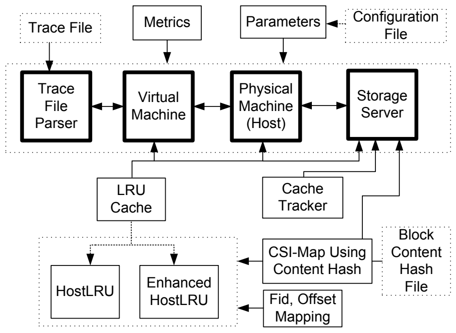

Fig. 4. Simulation framework to evaluate SeaCache.

host cache that can be configured to use either a *HostLRU* or *enhanced HostLRU* caching policy.

Storage Server: This main module models a storage server with an LRU cache, host-cache content tracker, and CSI-Map sharing features. It further uses the **Cache Tracker** module to keep track of content of host caches.

Support Modules: These modules facilitate the main modules. The **LRU Cache** module provides a content-based cache implementation that can be instantiated by the main modules as needed. The **Metrics** and **Parameters** modules are linked with all the main modules to enable flexible configuration, and produce different observable metrics. Specifically, the **Metrics** module keeps track of hit ratio, I/O bandwidth usage, latency of each request, total number of commands, number of read/write commands, number of logical blocks etc. The **Parameters** module takes charge of parsing the configuration file and setting up the corresponding module. Example parameters include cache size and policy, number of hosts, and number of storage server.

Finally, the configuration file provides an easy means for exploring the design space without modifying the source code. The modularity and flexibility of this framework greatly speeds up the simulation process.

B. Implementation

We have implemented a proof-of-concept read protocol prototype of SeaCache, specifically the CSI-Map sharing solution by modifying the NFS v3 protocol, client and server components in Linux 2.6.32.15, using about 1200 lines of C code.

The implementation setup comprises of Linux-based hosts running Oracle® Virtual Box 3.2.8 to provide client VMs. The clients run Windows XP SP3 with disks mounted via NFS. We use a write-through cache policy to ensure that we can use NFS v3 close-to-open consistency model. For computing CSI, we simply use the offset of the block and assume it to be a sufficiently unique content identifier. This is in-line with similar assumptions made in hypervisor design, which uses this concept to maintain a common base disk for multiple VMs by separating overwrites using snapshots for their VDI environments [14].

NFS Client: We trap *nfs_readpage(s)*, *nfs_readpage_result*, and *nfs_wb_page* NFS calls to enable CSI sharing and to service client block requests. The CSI-Map maintains two data-structures: (a) A *Fid-Offset* hashtable, which maps (file-handle, offset) to the actual PBNs; and (b) a *CSI hashtable*, which maps to a list of *Fid-Offset* entries that have the same content. The macro implementation of `uthash` [10] was used for the hashtable implementation. CSI-Map also supports an LRU list for removing (writing to disk) least-used entries if needed. The client *nfs3_xdr_readargs* and *nfs3_xdr_readres* RPCs are modified to marshal the SS-Push. Note that some of the data structures in CSI-Map have been built in anticipation of incorporating and integrating protocol information exchange with an NFS server that supports a de-duplicated file system, such as OpenDeDup [11].

NFSD Server: The NFS server maintains an exception list for all files opened by a particular NFS client. This list identifies block offsets that have been modified by any of the open files. This information is marshalled into a RPC to the client by modifying the *nfss3svc_decode_readargs* and *nfs3svc_encode_readres*.

We instrumented the Linux kernel to identify the cache hits and misses to our cache as well as the latency observed by each request. For testing, the clients ran typical OS operations such as booting, virus scan, and compilation of source code.

C. Workloads

In this section, we briefly describe the real-world traces that we have used to drive our evaluation of SeaCache.

CIFS Network Traces includes I/O traces collected over a period of four months from two large-scale enterprise storage servers deployed at a company which uses Common Internet File System (CIFS) as the network protocol and hosts about 1500 employees.

VDI Traces comprise of traces collected for two weeks from a system that was supporting 9 VMs in an in-house Virtual Desktop Environment. Here, in order to separate user-generated I/O from other accesses we disabled any anti-virus program on all the VMs. The VMs read 17.3 GB and write 6.1 GB data per day on average.

Due to resource consolidation efforts, in addition to exhibiting general usage characteristics, the VDI environment exhibits some interesting spikes in I/O requirements at certain times of the day. In VDE, login/boot storms are generated on storage boxes where a large spike of read requests are created as users login/boot to their desktop. Such a storm is highly predictable because most corporate users start using their computers around the same time, e.g., 9 am. In certain VDEs, the scheduled 3 am virus scan triggers a virus-scan storm. These are read intensive storms that could be mitigated as most of the clients request identical set of blocks from the storage server. Similarly, write intensive storms such as a patch update or virus update can be configured to occur in the trace periodically.

Test-Dev Trace: An enterprise level test-development environment uses resources continuously to 1) deploy/compile code, 2) install builds, and 3) perform QA activities. The QA integrated test environment often consists of a number of sub-environments where each sub-environment is associated with the testing of a particular feature being developed. Thus, at any given point in time, one of the above three processes are running for each of the QA sub-environments. We recreated such a test-dev environment for trace collection. In our setup, there were around 20 build-test VMs (each of which contained a sub-environment) on a physical host. Since most test-dev cycles are almost identical, we emulated a larger setup by replicating the traces and using simple Poisson arrival process to vary the start time of each instance of a test-dev cycle, finally giving us our Test-Dev trace. Within a single Test-Dev environment, while 40 MB data is read, 250 MB data is written. This is a write intensive trace.

V. Evaluation

We evaluate SeaCache using our simulator and our prototype implementation of Section IV-B. In our simulator, we configure the cache size of each VM to be 256 MB, each physical host to accommodate 10 VMs at most, and the cache size of storage server to be 4 GB. Most of our experiments measure the I/O bandwidth consumption between storage server and physical hosts, which are illustrated as **bars**, and the average latency of the I/O requests, which are illustrated as **stars** on the same graphs. In the following, we present the details of our experiments and observations.

A. CSI Sharing Protocol Analysis

In our first set of experiments, we use our simulator to analyze the different CSI sharing protocol algorithms described in Section III using the CIFS, VDI and Test-Dev workloads. The protocols being analyzed are as follows: 1) *Baseline* protocol that does not transfer any CSI information, and we use it as the baseline to compare to when reporting performance improvement results; 2) *Dedup-Read* approach used by deduplication box on read path; 3) *W/Tracker* where the storage server exactly tracks the host cache contents; 4) *SS-Push* where the storage server exactly tracks the host cache and pushes back more CSI to clients; 5) *Coop-Cache* using cooperative cache with SS-push; 6) *Dedup-RW* approach used by deduplication box both on read and write path; 7) *SeaCache* in which all the proposed features are enabled including read, write path and cooperative cache. For these experiments, the file system block size was set to $4KB$, and the network and disk latencies (obtained from real experiments) are modeled as 7.5 ms and 5 ms, respectively.

a) CIFS Trace: In this experiment, we analyze how much data is transferred between the storage server and 72 hosts for all of the above mentioned algorithms using CIFS trace. The trace involves 717 clients reading 17.2 GB and writing 7.3 GB of data. Figure 5 shows the I/O bandwidth consumption (in GB) and the average latency of CIFS trace. Note that, we consider the amount of data that will be exchanged between the hosts and the storage server as the measure of bandwidth consumption, as reducing this data results in better utilization of the available I/O bandwidth.

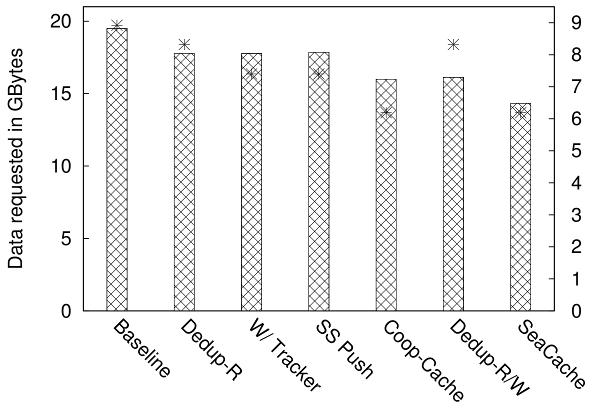

Fig. 5. I/O bandwidth consumption and average latency under CIFS trace.

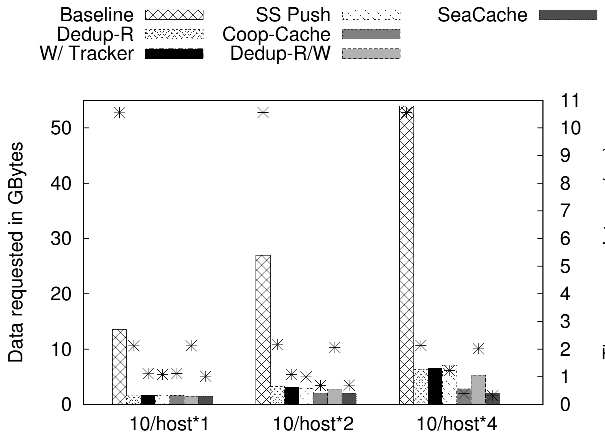

Fig. 6. I/O bandwidth consumption and average latency under virus-scan storm.

Baseline performs worse than the other algorithms by consuming 19.5 GB bandwidth with average latency of 8.92 ms because it does not share CSI information. *Dedup-Read* reduces I/O bandwidth consumption by 8.7% and lowers the average latency down to 8.6 ms by eliminating duplicate read requests. For our two read path protocol variants, *W/ Tracker* and *SS-push*, the total I/O bandwidth reduction is about the same with *Dedup-Read*, while the latency is reduced down to 7.4 ms. This shows our two variants can effectively remove the extra I/O consumed by *Dedup-Read*.

Dedup-RW reduces I/O bandwidth consumption by 17.3%, while *Coop-Cache* and *SeaCache* reduce I/O bandwidth 17.9% and 26.5%, respectively. We see that *SeaCache* outperforms *Dedup-RW* by 9.2% in terms of bandwidth reduction and 25% in terms of latency. This is because *SeaCache* effectively optimizes the read path and redistributes the read path workloads to other physical hosts.

b) Virus Scan in VDI Trace: Figure 6 shows the I/O bandwidth consumption and average latency in VDI environment for a Virus-Scan Storm under different protocols and different number of VMs. The three groups of bars in the graph are: 10 VMs on a single host, 20 VMs on two hosts, and 40 VMs on 4 hosts. Within each group, the I/O bandwidth

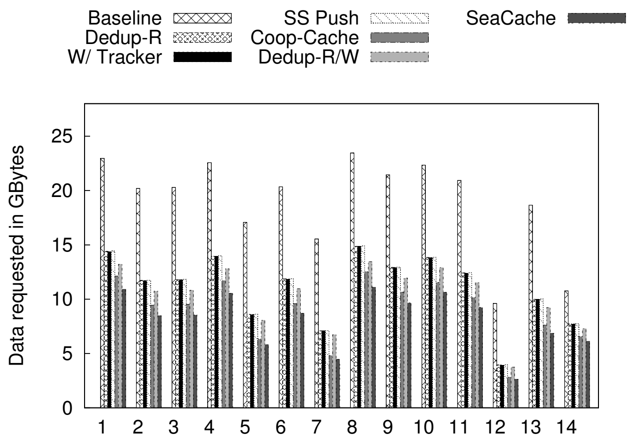

Fig. 7. I/O consumption on persistent VDI traces for two weeks of usage.

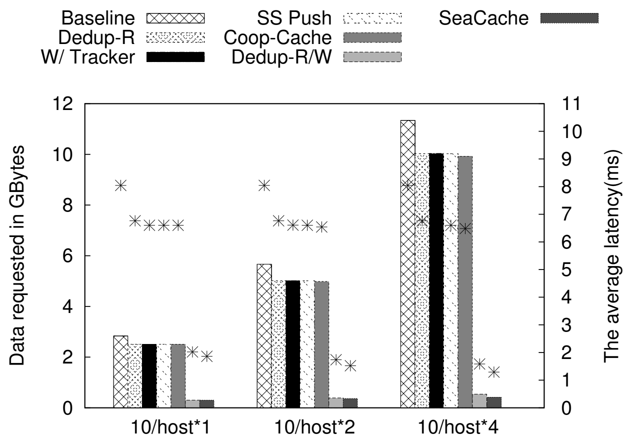

Fig. 9. I/O bandwidth consumption and access latency under Test-Dev traces.

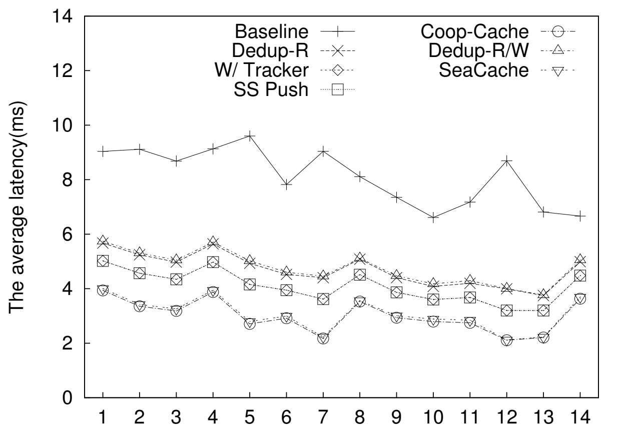

Fig. 8. Average latency per I/O on persistent VDI traces for two weeks of usage.

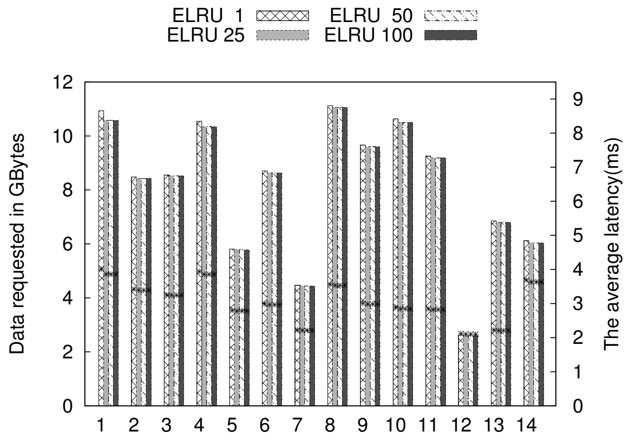

Fig. 10. Enhanced LRU in host cache for persistent VDI traces.

consumption and average latency per I/O are presented. It is observed that *SeaCache* is the best protocol compared with other six variants, as it achieves up to 96% I/O saving and 97% latency reduction compared with *Baseline*. The more clients are involved, the more benefits we can get from *SeaCache*. We can see that when 40 VMs are running, *SeaCache* outperforms *Dedup-RW* by 6% and 7% in corresponding I/O and latency reduction, respectively. Note that, even when the number of VMs running virus scan traces increases from 10 to 40, the I/O load seen by server increases by only $1.4\ X$ under *SeaCache*, while *Baseline* saw the load increase by $4\ X$.

c) Two Weeks VDI Trace: For this experiment, we traced the VDI environment for two weeks. Here since we have only 9 VMs involved, to show the effectiveness of *SeaCache*, we configure each physical machine to host at most 3 VMs. Thus, in this experiment, a total of 3 physical hosts are used.

Figure 7 and Figure 8 show I/O bandwidth consumption and average latency, respectively, under different scenarios for the VDI trace. The x-axis here represents each day of the two weeks trace duration. *Baseline* perform significantly worse compared to any of the CSI sharing algorithms in terms of the amount of data transferred between hosts and storage server as well as the latency. This experiment shows that the performance of *SeaCache* consistently exceeds *Dedup-RW* by up to 14% in I/O saving and 24% in latency reduction.

d) Test-Dev Trace: Similar to Figure 6, Figure 9 shows the I/O bandwidth consumption and average latency under different algorithms for the test-dev trace. The *Dedup-R* and our read path optimizations do not gain significant benefits because the test-dev trace is write intensive with write ratio of 86%. On the other hand *Dedup-R/W* and *SeaCache* can effectively reduce the duplicate writes. However, *SeaCache* does not win much over *Dedup-R/W* in this case since *SeaCache* focuses more on read path optimization (improving 1% in data saving and 3% in latency reduction). As in the previous experiment, increasing the number of VMs running the trace from 10 to 40 only results in a $1.8\ X$ increase in the I/O load seen by the storage server, where as *Baseline* experienced a $4\ X$ load increase.

The above experiments show that SeaCache can enable data center managers to provision storage servers for average loads instead of peak loads, and consequently improve the overall efficiency of the center.

B. Efficiency of Enhanced Host Cache

In our next experiment, we use the simulator to test whether enhanced LRU (ELRU) algorithm that takes CSI information

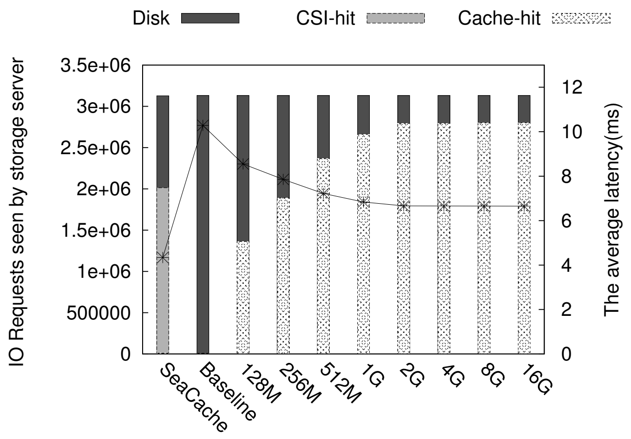

Fig. 11. Impact of CSI on storage server cache for data blocks.

into account can perform better in comparison to basic LRU for host cache management. Figure 10 shows the results for the two weeks of VDI traces under *SeaCache*. Here, ELRU-n implies that the least shared entry within the last n LRU entries is evicted, e.g., ELRU-1 is the same as LRU. It is observed that ELRU-n performs slightly better than LRU; improving 1.5% in bandwidth saving and 1.5% in latency reduction. The main reason for this behavior is that LRU already accounts for recent accesses. Thus, if a shared block is being accessed multiple times by different VMs, it is not evicted as it is often not the least recently used block.

C. Storage Cache Efficiency

In this experiment, we compare the impact of adding more memory to the storage server for storing data blocks (without CSI sharing protocol) with a storage server that uses our CSI sharing protocol but does not use extra cache. Note that the I/Os shown here are for data that is not present in the client cache, which are sent to the storage server. For this test, we pick the 5^{th} day of VDI traces, as that yields decent performance under *W/ Tracker*. As shown in Figure 11, *W/ Tracker* is able to perform better against the one with the extra cache (as much as 16 GB) for mitigating disk I/O. This is because any CSI-Map hit causes the storage server to respond to the host with just CSI information. In contrast, although any storage server data cache hit eliminates disk I/O, it does not prevent the transmission of the larger payload to the host. The key insight here is that the CSI sharing protocols reduce the size of payload that needs to be serviced back by the storage server, which drastically reduces the average latency experienced by the hosts. This is an important result because it emphasizes that the capital spent on provisioning larger caches on the storage server can now be moved to the hosts. Finally, more hosts can be served using a single storage server, especially for workloads that are friendly to our protocols, such as VDI or Test-Dev environments.

D. Implementation Results

In this experiment, we use our prototype implementation to show that CSI sharing is a feasible idea. To this end, we traced the I/O requests of booting Windows XP-SP3 one after the other 100 s apart. Each boot of VM requests about 400 MB of data.

TABLE II
AVERAGE LATENCY PER I/O ON VM BOOT.

	Network Hit (%)	CSI-Map Cache Hit (%)	Average Latency (ms)
Baseline VM1	100.00	-	8.58
Baseline VM2	100.00	-	7.29
SS-Push VM1	99.00	01.00	8.97
SS-Push VM2	64.56	35.44	3.39

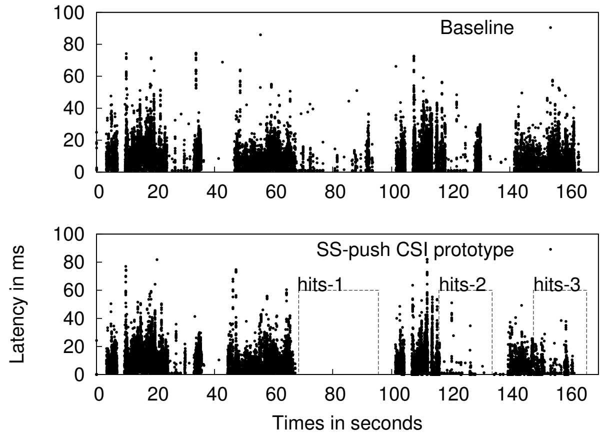

Fig. 12. Latency of each I/O request on booting two VM's one after another.

First, we compare the overhead introduced by CSI sharing in terms of average latency of booting the first VM for both *Baseline* and *SS-Push*. Table II shows that the overhead is negligible. Next, we observe the latency for booting the second VM. The CSI-Map cache hits drastically reduce the average latency of blocks requested on booting the second VM: about 35% of *SS-Push* requests for VM2 were identical to that for VM1, and these were cache hits in the CSI-Map. That shows that host can absorb the boot storm with out impacting the storage server, which would drastically reduce the design requirement on storage server from handling peak loads to average load.

Next, we measured the request latency for each I/O request as shown in Figure 12. We have marked three regions on the figure: hits-1, hits-2 and hits-3. Hits-1 presents the region where the *SS-Push* for VM1 re-requests blocks again post boot. These requests are absorbed by the CSI-Map cache and therefore we observe no latency in servicing these requests. The regions hits-2 and hits-3 mark requests by *SS-Push* for VM2 that has not been modified by VM1. We observed that the average latency for block-reads for VM2 dropped by 62%. With better CSI-Map between blocks and content, we expect to obtain a high rate of hits on a boot workload. This approach is very useful when tens of VM's are booted on a host, and the approach can mitigate boot-storms or virus-scan storms using a software-only solution.

VI. Conclusion

In this paper, we presented an integrated approach, SeaCache, which incorporates host side caching, storage server to host data transfer and de-duplication information sharing protocols, and a storage server side de-duplication mechanism. SeaCache allows data center operators to (a) not provision resources for peak loads (for VDI type workloads) and (b) not procure extra hardware resources such as caches or on-wire de-duplication boxes. In this regard, we present: 1) algorithms for how storage server side de-duplication information can be leveraged to optimize storage server to host data transfers and host side caching; and 2) how host side client cache information can be leveraged at the storage servers to efficiently perform data transfer operations. We analyzed the proposed algorithms using three real-world workload traces and the results support our hypothesis that looking at these three system design areas in an integrated manner leads to overall bandwidth and latency benefits. Our experiments show that compared to dedup-box, SeaCache improves the I/O saving by up to 14% and latency reduction by up to 25%. Moreover, the results prove that SeaCache effectively absorbs the peak load under Virus-scan storm and Test-Dev traces. We have also developed a *proof-of-concept* implementation of SeaCache with an NFS client using a modified NFS protocol (that is CSI sharing protocol aware) and made the necessary changes at the NFS server to further validate that the approach is viable.

Acknowledgment

This paper is based upon work supported in part by the National Science Foundation under Grants CCF-0746832, CNS-1016793, and CNS-1016408.

References

[1] F. 180-1. *Secure Hash Standard*. U.S. N.I.S.T. National Technical Information Service, Springfield, VA, Apr. 1995.

[2] D. C. Anderson, J. S. Chase, S. Gadde, A. J. Gallatin, K. G. Yocum, and M. J. Feeley. Cheating the I/O bottleneck: network storage with trapeze/myrinet. In *Proc. USENIX ATC*, 1998.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, 2005.

[4] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for RAIDs. In *Proc. 31st ACM/IEEE ISCA*, 2004.

[5] A. T. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proc. USENIX ATC*, 2000.

[6] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36(SI):285–298, 2002.

[7] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proc. 1st USENIX OSDI*, 1994.

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: harnessing memory redundancy in virtual machines. In *Proc. 8th USENIX OSDI*, 2008.

[9] J. Hamilton and E. W. Olsen. Design and implementation of a storage repository using commonality factoring. In *Proc. IEEE MSST*, pages 178–182, 2003.

[10] T. D. Hanson. http://uthash.sourceforge.net/. Uthash: A hashtable for C structures. Sept. 2010.

[11] http://www.opendedup.org. Open dedup, Sept. 2010.

[12] N. Jain, M. Dahlin, and R. Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proc. 4th USENIX FAST*, pages 21–21, Berkeley, CA, USA, 2005.

[13] S. Jiang, F. Petrini, X. Ding, and X. Zhang. A locality-aware cooperative cache management protocol to improve network file system performance. In *Proc. IEEE ICDCS* , pages 42–49, Washington, DC, USA, 2006.

[14] S. Jin. *VMware VI and vSphere SDK: Managing the VMware Infrastructure and vSphere*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2009.

[15] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve i/o performance. In *Proc. 8th USENIX FAST*, 2010.

[16] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proc. USENIX ATC*, pages 5–18, Berkeley, CA, USA, 2004.

[17] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-tier cache management using write hints. In *Proc. 4th USENIX FAST*, pages 9–25, Berkeley, CA, USA, 2005.

[18] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. 7th USENIX FAST*, pages 111–123, Berkeley, CA, USA, 2009.

[19] P. Macko, M. Seltzer, and K. A. Smith. Tracking Back References in a Write-Anywhere File System. In *Proc. 9th USENIX FAST*, Berkeley, CA, USA, 2010.

[20] G. Milos, D. G. Murray, S. Hand, and M. Fetterman. Satori: Enlightened page sharing. In *Proc. USENIX ATC*, 2009.

[21] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. 8th ACM SOSP*, pages 174–187, New York, NY, USA, 2001.

[22] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron. Everest: scaling down peak loads through I/O off-loading. In *Proc. 8th USENIX OSDI*, pages 15–28, Berkeley, CA, USA, 2008.

[23] A. Osuna and R. F. Javier. *IBM System Storage N series Software Guide*. IBM Redbook, July 2010. SG24-7129-04.

[24] S. Quinlan and S. Dorward. Venti: A New Approach to Archival Data Storage. In *Proc. 1st USENIX FAST*, pages 7–20, Berkeley, CA, USA, 2002.

[25] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. USENIX ATC*, pages 143–156, Berkeley, CA, USA, 2008.

[26] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: recapitulating storage for virtual desktops. In *Proc. 9th USENIX FAST*, pages 3–16, Berkeley, CA, USA, 2011.

[27] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, B. Thomas, and A. Perrig. Opportunistic use of content addressable storage for distributed file sytems. In *Proc. USENIX ATC*, 2003.

[28] C. A. Waldspurger. Memory resource management in Vmware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[29] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proc. USENIX ATC*, pages 161–175, Berkeley, CA, USA, 2002.

[30] Y. Zhang, N. Ansari, M. Wu, and H. Yu. On wide area network optimization. *IEEE Communications Surveys Tutorials*, PP(99):1–24, 2011.

[31] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. 6th USENIX FAST*, pages 1–14, Berkeley, CA, USA, 2008.