Host-side Filesystem Journaling for Durable Shared Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis
Department of Computer Science and Engineering, University of Ioannina, Ioannina 45110, Greece

Abstract

Hardware consolidation in the datacenter occasionally leads to scalability bottlenecks due to the heavy utilization of critical resources, such as the shared network bandwidth. Host-side caching on durable media is already applied at the block level in order to reduce the load of the storage backend. However, block-level caching is often criticized for added overhead, and restricted data sharing across different hosts. During client crashes, writeback caching can also lead to unrecoverable loss of written data that was previously acknowledged as stable. We improve the durability of shared storage in the datacenter by supporting journaling at the kernel-level client of an object-based distributed filesystem. Storage virtualization at the file interface achieves clear consistency semantics across data and metadata blocks, supports native file sharing between clients over the same or different hosts, and provides flexible configuration of the time period during which the data is durably staged at the host side. Over a prototype implementation that we developed, we experimentally demonstrate improved performance up to 58% for specific durability guarantees, and reduced network and disk bandwidth at the storage servers by up to 42% and 82%, respectively.

1 Introduction

Infrastructure virtualization in the datacenter typically consolidates client and server nodes on similar hardware. Network storage is often provided by scalable server clusters through protocols operating at the file, block or object level. The file interface is attractive for its sharing and efficiency properties [30, 13, 25, 40, 6, 22, 1]; the block interface provides convenient virtualization flexibility but incurs undesirable translation overheads [24, 20, 35, 36, 26]; and the object interface is scalable and efficient because it carries semantical information for specialized storage management [30, 39, 41].

Another design dimension in datacenter storage applies client-side caching for improved performance and durability at reduced network and server load. Existing solutions often apply block-level caching at the client-side host, and they adopt write-through or writeback policy according to the application and hardware characteristics. A write-through policy is preferred for read caching without data loss at device failure. Instead, a writeback policy improves the resource efficiency and application performance but makes the cache device part of the failure model [24, 34, 9, 19, 31, 16].

The Arion system is a new design point that we introduce in cloud storage to improve the durability of the file interface at the client side (Fig. 1 fully explained in Section 2). We

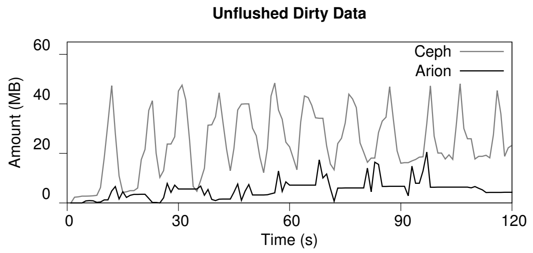

Figure 1: Dirty data that remains unflushed in the volatile memory of the client in Ceph and the proposed Arion system.

integrate the client software of a distributed filesystem with persistent host-based storage over a journal device. We enhance the flushing functionality of the filesystem client with tunable control of both the amount of dirty pages that are staged at the host, and the time period taken by dirtied pages to reach the backend servers. At increased flushing frequency to the journal device, we practically minimize the recovery point objective (RPO) close to zero under the following condition: the dominant cause of a client crash is operator or software bug rather than permanent hardware loss, such as that of the local storage device [28, 4].

The availability and performance of cloud services depends on the ordering, durability and membership properties of replication consistency [7]. In a multi-tier system, data is replicated at the frontend application, an intermediate caching layer, and the backend persistent storage. For reduced cross-layer communication, the frontend can be stateless and lose recently written data during a crash or reboot. Recognition of this risk has urged the designers of local filesystems, flash-based caches and distributed storage systems to emphasize the ordering guarantees of crash consistency at the expense of weaker durability [10, 19, 26].

I/O-intensive workloads in a distributed filesystem can take advantage of writeback caching at the client side to improve their performance and reduce the respective network and server load. Unfortunately, current scalable filesystems can natively support only in-memory caching at the client side. This deficiency has been partially addressed by having the filesystem client running in the hypervisor and enforcing the guests to mount disk images as plain files through a block interface that enables block-based caching [9]. But this approach has been criticized for the increased overheads from the semantic gap that it causes and the unnecessary multiple translations between the file and block interface [13, 20, 35].

Our main contributions are the following: (i) We improve the durability of frontend memory caching by integrating disk-based journaling into the client of a distributed filesystem. (ii) We implement a prototype of the proposed storage layer in the kernel-level client of the Ceph object-based filesystem. (iii) We experiment with several application-level benchmarks over a virtualized host and a clustered storage backend. Our approach enables frequent flushes of dirty pages to the local journal without crossing the network and hitting the disks of the backend storage. Therefore, in a host machine with reliable local storage, we approximate the consistency ordering and durability of write-through caching with the configurable efficiency of periodic writeback.

We further motivate our research in Section 2 and describe the Arion architecture in Section 3. We present our software prototype in Section 4 and explain our experimental results in Section 5. In Sections 6 we compare our work with previous research, and in Section 7 we clarify our contributions and some limitations. Finally, in Section 8 we outline our conclusions and possible extensions.

2 Motivation

Loss or corruption of committed updates to critical data is recognized as a particularly damaging class of failure [4]. This observation is highly relevant in a large-scale multi-tier environment, with mean time between failures inversely proportional to the number of machines. Several studies conclude that hardware failures contribute much less to service-level failures in comparison to causes related to software bugs and faults from operator or maintenance tasks [28, 4].

In traditional Unix, written data is acknowledged asynchronously to the application but only flushed periodically to the local disk. This approach has been adopted by several distributed filesystems in the form of asynchronous data transfer from the volatile memory of the client to the servers [27, 23]. Although durable caching at the client side can reduce the network load of the servers, it complicates the maintenance of replication consistency among different clients or between the clients and the servers [14].

In Fig. 1, we measure the amount of dirty data that remains unflushed at the client memory over time. We compare Ceph [41] under default flushing parameters with the proposed Arion system (Section 3). In the environment of Section 5, we used the fileserver mode of Filebench [12] running for 2min over 10000 files. The Linux pdflush daemon wakes up every 5s and transfers dirty data older than 30s from the client to the servers [8]. Additionally, the Arion client every 1s flushes dirty data to the local journal of the host. On average over time, the Ceph client keeps 24.3MB of dirty data solely in volatile memory, i.e., unrecoverable from a crash. Instead, the Arion host-side journaling reduces to 5.4MB the vulnerable data in the volatile memory of the client.

3 System Architecture

Next we outline our assumptions and goals before we describe the main design ideas of Arion and consistency.

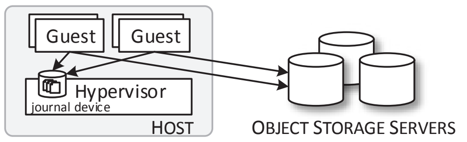

Figure 2: Host-side journaling in the Arion architecture.

3.1 Assumptions and Goals

We aim to improve the durability and performance of shared storage in the datacenter at reduced utilization of the server resources. User is the application-level entity that initiates I/O requests to the filesystem, and client is the host-based software that provides filesystem access to users. We target host hardware with reliability characteristics on par with those of the server machines. The host provides directly-attached storage with sufficient redundancy to tolerate the occasional failure of a single device. Appropriate storage technologies include hard disks, solid-state drives, or non-volatile memory. In the proposed storage architecture we aim to support the following properties:

i) **Interface** Stored data is directly accessible for regular use and maintenance tasks over the network with a POSIX-like file-based interface [37].

ii) **Sharing** Heterogeneous clients on the same or different hosts can natively share data at the storage level but may also apply synchronizations at the application level.

iii) **Durability** Most recent writes survive client reboots but require redundant hardware support to tolerate permanent failures of individual storage devices at the host.

iv) **Performance** Client writes are safely stored at sequential disk throughput, but the read performance depends on the efficiency of the client memory cache.

v) **Scalability** The storage backend linearly scales out to efficiently hold increasing amounts of data.

3.2 Design

We rely on an object-based scale-out backend of multiple data and metadata servers (Fig. 2). The client runs over either a guest system on virtualized hardware or a standalone system on bare metal. A read operation synchronously returns the latest version of the requested state. A synchronous write reaches a configurable number of durable replicas before it returns. An asynchronous write returns as soon it updates the buffer cache of the client system, but the modified blocks have to reach a configurable number of durable replicas before they are considered safely stored.

We regard the frontend logging to a persistent storage medium as a complementary form of replication. Unlike the traditional replication that is homogeneously applied across functionally equivalent backend servers, the frontend logging adds *heterogeneity* with respect to the storage format, the logical layer and the time duration of the replica[1].

[1]The Coda filesystem previously introduced the concept of two-tier replication in the context of disconnected operation [18](see also Section 6).

The metadata server (MDS) enables shared filesystem access at *file* granularity through different types of tokens leased to the clients. Supported access types include exclusive write (cached) by a single client, and concurrent read (cached) or concurrent write (uncached) by multiple clients. A client can only cache the writes of data and metadata accessed with exclusive permission. The file interface provides valuable semantical information about the consistency dependencies of modified data and metadata. When a client transfers the file updates to the servers, the metadata is written only after the referenced data blocks have safely reached the server state.

The key innovation of our design is the integration of a local journal with the kernel-level client of a distributed filesystem (Fig. 2). The host-side journal is distinct for each guest in virtualized hosts. The client inserts into the journal *both* the data and metadata modified by an I/O request. Thus we ensure that a metadata version matches the version of the data it refers to (version consistency [11]). We only keep one transaction active to accept all the (redo) records of low-level I/O operations corresponding to an atomic filesystem request. An active transaction closes as a result of timeout expiration, explicit flush request, or reclamation of journal space [8].

A journaled block remains cached in the client memory until it is safely written to the servers. If an MDS revokes the write token from a client due to some conflict (e.g., concurrent writes to the same file), the client is forced to write (checkpoint) the conflicting writes to the servers and invalidate the respective journal records. On client disconnection from the servers, the leased tokens may expire and the client will no longer be able to access the files locally [23]. At network reconnection, the client writes to the servers the mutated blocks of each file whose token has been refreshed and whose metadata cached at the client is newer than the file metadata at the MDS, but it discards the remaining blocks.

The primary benefit from host-side journaling in a distributed filesystem is the reduced vulnerability of outstanding writes in the volatile memory of the client. If a client crashes and reboots without hardware failure at the host, then the client replays the completed transactions and transfers the recorded updates to the filesystem servers. We update a file only if the replaying client confirms token ownership, and the journaled metadata is newer than the file metadata at the MDS. In case of client crash during the recovery, the replay is repeated until the client journal is fully checkpointed.

The durable storage of recent writes over the host-side journal improves the server writeback efficiency with respect to the utilized network and disk bandwidth. The consumed shared resources are reduced through batching applied to repetitive writes over the same blocks, or to small writes. At synchronous writes, we journal the updates locally and postpone the server writeback as permitted by the flushing parameter settings. Thus, performance improves depending on the pressure over the shared resources and the resulting queuing delays in the I/O path of the Arion networked storage.

3.3 Consistency

We strengthen the durability of memory-based caching in clients that provide native support for file sharing. The file interface differentiates the data blocks from the metadata. Thus, our system cleanly addresses issues of vertical (client-server) and horizontal (client-client) consistency across different replicas [9]. In the order imposed by their arrival time and structural dependencies, the data and metadata updates are first journaled at the local host and subsequently persisted at the backend servers. Additionally, the filesystem arbitrates the conflicts among different clients through lease-based tokens. In contrast, block-based schemes typically operate transparently to the filesystem, and as a result explicitly track the order and *relax* the durability of block updates [10, 19, 26].

4 System Prototype

Next we provide background information on the Linux kernel and Ceph, before we present the implementation of Arion.

4.1 Background

Linux The Linux kernel maintains in memory a page cache with data and metadata blocks of recently accessed disk files [8]. A page descriptor stores bookkeeping information about the address space and the inode of a page. For every cached disk block, there is a block buffer that stores the actual data, and a buffer head structure with bookkeeping information. The dirty pages are written to disk at timeout expiration, under space pressure in the main memory or the journal device, and by explicit flush request from the user.

The Linux kernel implements filesystem journaling with a special kernel layer, the Journaling Block Device (JBD). All the records of the low-level operations that belong to a high-level atomic update are stored in the same transaction of the journal. The journaling I/O of each block buffer is managed through a separate buffer head structure in kernel. Additionally, a journal head structure links each block buffer to the corresponding transaction. One or more journal descriptor blocks mark the beginning of the transaction and store the tags of journal blocks belonging to the transaction. A commit operation writes to the journal the dirty buffers of a transaction followed by a commit block. A checkpoint operation transfers the records of a transaction to the filesystem state and deletes the transaction from the journal.

Ceph The Ceph is an object-based parallel filesystem designed for scalability, performance and high availability [41]. It consists of four main components: the clients provide a POSIX-like I/O interface; the metadata servers (MDS) manage the namespace hierarchy; the object storage devices (OSD) reliably store data and metadata; and the monitors (MON) manage the server cluster map. A set of MDSs acts as a scalable, consistent, distributed cache of the file namespace. The metadata is persistently stored on the OSDs as a collection of regular objects. Ceph maps each object to a placement group consisting of multiple OSDs. Each

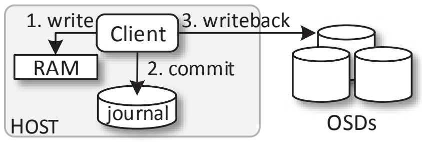

Figure 3: A write request is applied to the kernel memory, then added to the host journal and finally reaches the servers.

OSD maintains a local journal to handle object versioning and update serialization.

The kernel-level client prepares a page in the cache upon a write request. A partial page update fetches the original page to the cache from the OSDs, copies on it the user modifications and marks the inode object as dirty. The Linux pdflush threads wake up periodically to scan the list of dirty inodes and write their dirty pages to the OSDs. The writeback time refers to the wake-up period, and the expiration time refers to the time length after which a dirty page is flushed. After the writeback completion at the OSDs, the client transfers the dirty inode to the MDS and receives acknowledgment when the inode update is safely stored.

4.2 Implementation

In order to improve the durability and efficiency of shared data storage over Ceph, we increase the statefulness of the filesystem client with a local journal (Fig. 3). Our prototype implementation integrates the Linux JBD with the CephFS kernel-level client and the other Ceph components. In the `ceph_fs_client` structure of CephFS we added two extra fields: the `journal_bdev` referring to a block-device control structure in the kernel, and the `s_journal` referring to the journal control structure of JBD. We specify the journal device to the kernel with a new mount option that we added in the system.

For the addition of journaling support in Ceph, we allocate disk block buffers and buffer heads during a write with the `ceph_write_begin()` function. We also create a journal head to insert each block buffer to an active transaction. The modified pages are written to the network servers either periodically, or under pressure from the client memory and journal space. After a particular page is written back or invalidated, we also invalidate the respective journal records.

The `private` field of a page descriptor is used by the local Linux filesystem to typically link a page with a block buffer, and by Ceph to maintain the context of the supported snapshot service. Instead, we introduce an auxiliary metadata structure, called `ceph_metapage`, to associate a page with the block buffer, the snapshot context, and several inode attributes of the file. We link the above metadata structure to the page via the `private` field of the page descriptor. Arion allocates the `ceph_metapage` structure when it creates the block buffers for a page, and deallocates it after the page is written back to the servers.

In kernel, we added a new page state, called `JBD_state`, to indicate that a page has been marked for journaling but not committed yet to the journal. The journaling of metadata operations has been particularly challenging to implement in Arion, because there are several places in the I/O path of the original Ceph that mark the inode as dirty. Also, unlike local filesystems, the Ceph client does not directly cache an inode object as a raw metadata block. In order to effectively manage the file metadata in the journal of Arion, we substantially expanded the JBD tag structure in the journal descriptor block to include several inode attributes provided by the MDS.

As a result, the journal tag of Arion contains fields to identify the number of modified data blocks, the modification offset range, and the inode number, version, size, permissions and latest time of different operation types. During the crash recovery of a client, we compare the inode metadata contained in the journal tag of a file against the respective attributes freshly fetched from the MDS. Subsequently, we only replay the write requests of files whose journaled metadata has not been obsoleted in the MDS by accesses that occurred in the time period between the transaction commit and the ongoing recovery.

Our current prototype implementation fully supports (i) the journaling of mutated data and metadata from the client memory to the host-side journal, and (ii) the filesystem recovery after a client crash that leaves the host hardware operational.

5 Performance Evaluation

We implemented the Arion host-side journaling based on Linux JBD2 and the kernel-level client of Ceph (v0.80.1). The Arion development required 3417 new commented lines across 15 files of Linux kernel (v3.6.6). Next we describe our experimentation environment, and the measured performance and resource consumption of Arion and original Ceph.

5.1 Experimentation Environment

The host machine is a rack server with 2 quad-core x86-64 2.66GHz processors, 7GB RAM, 2 bonded 1GbE links, and two 300GB 15KRPM SAS HDDs in RAID0 configuration. The host uses Linux kernel v3.5.5 with Xen v4.2.0, and the guest runs Linux v3.6.6 over 2GB RAM and 2 pinned VCPUs. Arion uses a 2GB disk partition at the host for local journal. We leave for future work the study with other types of durable devices (e.g., SSDs). The guest client mounts directly the distributed filesystem, and the hypervisor provides local access to the network and journal devices.

Each of Ceph and Arion uses 5 machines: 3 OSDs, 1 MON and 1 MDS. The machine is a rack server running Linux kernel (v3.10.41) over 2 quad-core x86-64 2.66GHz processors, 3GB RAM, 1 GbE link, and two separate 300GB 15KRPM SAS HDDs. A stored object is replicated over 3 OSDs. Each OSD dedicates one disk for journaling (1GB partition).

Our experiments are based on the Filebench v1.4.9.1 macrobenchmark (fileserver, varmail, createfiles) and the FIO v2.1.7 microbenchmark. We clear the caches before each experiment. We keep the on-disk write buffers *disabled* at

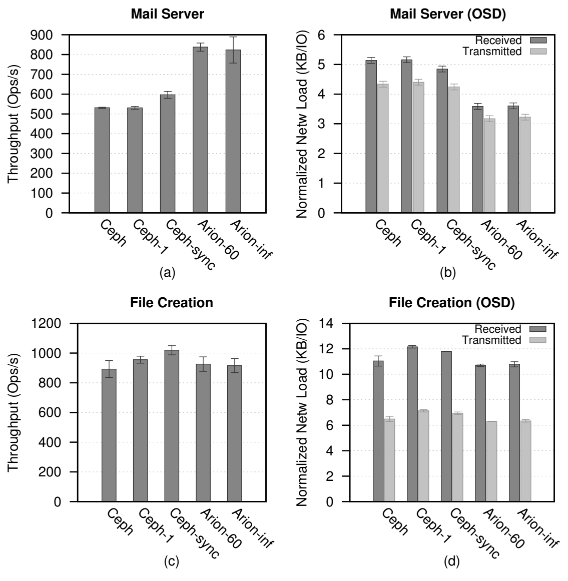

Figure 4: Operation throughput and normalized network load with the varmail (a,b) and createfiles (c,d) modes of Filebench across different settings of Ceph and Arion.

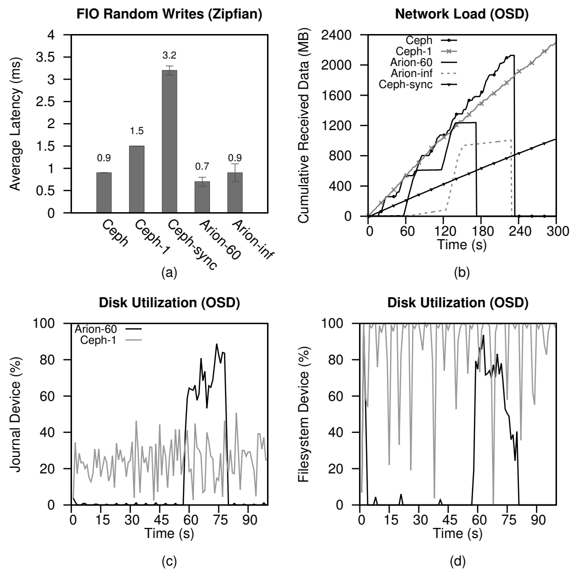

Figure 5: (a) Average latency, (b) cumulative network load at one OSD, and disk utilization at the (c) journal and (d) filesystem OSD disks across different settings of Ceph and Arion.

the host, but activated at the servers of Ceph and Arion [32]. RAID0 with two disks does not give unfair advantage to host journaling because the storage backend already consists of multiple servers with two disks each. In the shown bar charts we include 95% confidence intervals from 5 repetitions.

5.2 Measurements

In Fig. 4 we run two Filebench modes with default settings (for 5min) to study the system performance and efficiency. We examine Ceph with the writeback and expiration time respectively set to the default 5s and 30s (Ceph), or both set equal to 1s (Ceph-1), or the filesystem mounted in synchronous mode (Ceph-sync). We also examine Arion with dirty blocks periodically copied to the host-side journal every 1s, and the writeback and expiration times both set equal to 60s (Arion-60), or infinity (Arion-inf) to minimize writeback.

Varmail emulates multi-threaded I/O activity of a server *synchronously* storing email messages across 50000 files. In Fig. 4a, Arion-60 achieves operation throughput of 837.8 ops/s, or 58% higher than 531.3 ops/s of default Ceph. Arion-60 increases the Ceph data throughput (1.9MB/s) by 58% and reduces the Ceph latency (97.5ms) by 39%. The performance of Ceph-1 is similar to that of Ceph. Fig. 4b illustrates the received and transmitted OSD network traffic normalized by the number of completed operations. Arion-60 reduces the received network load of Ceph —normalized in KB/IO— by 30% and the transmitted by 27%. The bottleneck resource is the server disk I/O caused by synchronous writes.

We examine a metadata-intensive workload with file creations in Figures 4c,d. Arion-60 is comparable to Ceph with respect to performance and load. Ceph-sync achieves higher performance than Arion-60 by 10%, but also increases by 10% the received and transmitted network load. Ceph-sync improves slightly the performance because it handles the metadata updates directly at the MDS instead of fetching them to the client as asynchronous settings do.

We further explore the relative behavior of the two systems using the FIO microbenchmark with Zipfian write pattern of α=1.0001 (e.g., [17]). The benchmark writes a total of 2GB data in a preallocated file of 2GB size with block size in the range 2-16KB. With respect to Ceph (0.9ms) and Ceph-1 (1.5ms) in Fig 5a, Arion-60 achieves lower latency (0.7ms) by 22% and 53%, respectively. We examine the total network traffic received over time at one OSD of each system in Fig 5b. We notice that Ceph terminates at instance 233s with 2.1GB total received traffic. In contrast, Arion-60 ends the experiment at 172s (26% shorter) with received volume 1.2GB (42% lower).

In Figures 5c,d we examine the bandwidth utilization of the journal and filesystem storage device at one of the OSDs. We show Ceph-1 that keeps the durability characteristics similar to those of Arion. The depicted Arion-60 OSD utilizes the journal and filesystem device at 15.0% and 16.4% on average; the respective utilizations of Ceph-1 are 24.4% and 88.4%. We conclude that Arion-60 reduces the filesystem device utilization by 82% with respect to Ceph-1 in the examined case.

Overall, Arion-60 improves the performance of Ceph and Ceph-1 by up to 58%, but also reduces the server network and disk load by up to 42% and 82%, respectively. We experimentally confirmed the improved comparative performance and efficiency of Arion in several other write-intensive workloads (e.g., OLTP, key-value store). We also measured the recovery time of the Arion client in the range 77.4ms-2.622s, depending on the load of client write activity before the crash.

6 Related Work

Filesystems Andrew pioneered client disk-based caching but lacked the explicit separation in data and metadata management of object-based storage [33, 41]. Coda exploited data caching strictly for availability during disconnected operation [18]. During a communication failure, a Coda client logged locally the mutating system calls. At network reconnection, each server received and replayed all the logged operations together as one transaction. On the contrary, Arion continuously logs mutations during normal operation and writes them back to efficiently maintain consistency.

Database consistency can be preserved through transaction correctness [5]. SiloR is a multicore database system that uses logging and checkpointing for fast recovery to a transactionally-consistent state without replication [42]. The Sprite distributed filesystem disabled client caching of files concurrently updated by different clients [27]. Echo introduced ordered write-behind to delay the automatic writing of cached blocks to server disks [23]. NFSv4 delegates request handling to the client for reduced latency and network traffic [29]. Unlike Arion, existing filesystems typically limit client caching to volatile memory without support for durable host-side journaling during normal operation.

Virtualization and cloud VMFS stores disk volumes over shared cluster-based block storage [38]. Capo uses the local disks of the hosts for multicast-based preload and block-based write-through or writeback caching [34]. Ventana combines file-based sharing with the versioning, migration and access control of virtual disks [30]. A client-side manager offers disk-based caching but relies on NFSv3 at the host to connect the virtual machines with object-based storage servers. Therefore, existing storage systems only support block-level caching, or inherit the limited scalability of NFSv3.

CacheFS supports local disk-based caching but is practically limited to read-only filesystems [15]. BlueSky provides on-site NFS-based proxy service of remote cloud storage through local disk caching of journal and log segments [40]. SCFS provides FUSE-based caching of entire files at the client memory and disk without the proxy bottleneck [6]. However, it lacks the journaling integration with a scalable distributed filesystem of Arion for flexible file sharing.

Flash memory Non-volatile memory can be used at the client and server of a distributed filesystem for I/O efficiency [3]. Writeback caching can improve performance, reduce server load, and eliminate cache warmup on restart [2]. Optimistic crash consistency decouples ordering from durability for efficient filesystem consistency [10]. In-place commit over non-volatile memory unifies the buffer cache with journaling [21]. Offering disk-based caching through journaling is an extension of Arion that we plan for future work.

Mercury pointed out the zero recovery point objective (RPO), i.e., no recently-written data lost from a crash. It uses flash memory in the block I/O virtualization stack of the hypervisor to provide write-through caching [9]. Non-zero RPO can be applied for improved performance via block-level writeback caching at the host. Update order is preserved by explicit tracking of the dependency between I/O requests or transaction grouping of modified blocks [19]. Due to concerns about the consistency and durability of these ordering schemes, a recent block-level solution satisfies asynchronously but explicitly the ordering constraints of application-specified write barriers [31]. Nevertheless, host-side block-based caching lacks native support for writable file sharing within or across hosts [9, 19, 31, 2, 16].

7 Discussion

Persistent host-side caching primarily targets the improved performance and efficiency of networked storage. Typically, it uses a block-based interface that inherently lacks both the support for data sharing across different hosts and the ability for interposition in the file-based protocol of a distributed filesystem. It also makes the consistency preservation of network storage a challenging problem because the semantic gap between the file and block interfaces complicates the atomic grouping of dirty blocks by I/O request, and their ordering according to filesystem-imposed dependencies. Finally, the persistence of mapping metadata in block-based caching and the repetitive translation of I/O requests across different storage layers can introduce considerable overheads in networked storage I/O [2, 13].

The original design of Ceph cannot recover any writes that returned after they were only placed at the volatile memory of the client before a crash. Therefore, the Arion architecture is innovative because it adds durability into the client memory cache through journal-based recovery, conditionally propagates the updates to the servers after client reconnection, and also permits the clients to scalably communicate directly with the object servers of the storage backend. Overall, assuming host machines with sufficiently reliable local storage, our approach overcomes several sharing, scalability, and consistency limitations of related existing solutions.

8 Conclusions

For enhanced end-to-end durability of shared storage in the datacenter, we integrate the client of a distributed filesystem with a host-based journal. At the host, we provide local durable storage to dirty data and metadata until they are written to the network servers. We implemented a prototype of the proposed Arion design over the Ceph production distributed filesystem. In a virtualization environment, we experimentally demonstrate promising efficiency and performance results for specific durability levels configured through the frequency of copying dirty blocks to the host-side journal. In our future work we plan to experiment with different types of storage devices; explore interesting tradeoffs among performance, durability and efficiency for demanding applications; and extend the host-based journaling to support caching of blocks evicted from memory.

9 Acknowledgments

We thank our shepherd Jens Teubner for his valuable guidance and the anonymous reviewers for their constructive comments. This research has been supported by the European Union (European Social Fund - ESF) and Greek national funds through the Operational Program ”Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) — Research Funding Program: Thales. Investing in knowledge society through the European Social Fund (project Cloud9).

References

[1] APPUSWAMY, R., LEGTCHENKO, S., AND ROWSTRON, A. Towards paravirtualized network file systems. In *USENIX Workshop on Hot Topics in Storage and File Systems* (Philadelphia, PA, June 2014).

[2] ARTEAGA, D., AND ZHAO, M. Client-side flash caching for cloud systems. In *ACM Intl. Systems and Storage Conf.* (Haifa, Israel, June 2014), pp. 7:1–7:11.

[3] BAKER, M., ASAMI, S., DEPRIT, E., OUSTERHOUT, J., AND SELTZER, M. Non-volatile memory for fast, reliable file systems. In *ACM ASPLOS Conf.* (Boston, MA, Oct. 1992), pp. 10–22.

[4] BARROSO, L., CLIDARAS, J., AND HÖLZLE, U. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[5] BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Co., Reading, MA, 1987.

[6] BESSANI, A., MENDES, R., OLIVEIRA, T., NEVES, N., CORREIA, M., PASIN, M., AND VERISSIMO, P. SCFS: a shared cloud-backed file system. In *USENIX Annual Technical Conf.* (Philadelphia, PA, 2014), pp. 169–180.

[7] BIRMAN, K., FREEDMAN, D., HUANG, Q., AND DOWELL, P. Overcoming CAP with consistent soft-state replication. *Computer 45*, 2 (Feb. 2012), 50–58.

[8] BOVET, D. P., AND CESATI, M. *Understanding the Linux Kernel*, third ed. O'Reilly Media, Sebastopol, CA, Nov. 2005.

[9] BYAN, S., LENTINI, J., MADAN, A., PABON, L., CONDICT, M., KIMMEL, J., KLEIMAN, S., SMALL, C., AND STORER, M. Mercury: Host-side flash caching for the data center. In *IEEE Intl. Conf. on Mass Storage Systems and Technology* (Pacific Grove, CA, Apr. 2012).

[10] CHIDAMBARAM, V., PILLAI, T. S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Optimistic crash consistency. In *ACM Symp. on Operating Systems Principles* (Farminton, PA, Nov. 2013), pp. 228–243.

[11] CHIDAMBARAM, V., SHARMA, T., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Consistency without ordering. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 73–86.

[12] http://sourceforge.net/projects/filebench/.

[13] HILDERBRAND, D., POVZNER, A., TEWARI, R., AND TARASOV, V. Revisiting the storage stack in virtualized NAS environments. In *USENIX Workshop on I/O Virtualization* (Portland, OR, June 2011).

[14] HOWARD, J. H., KAZAR, M. L., MENEES, S. G., NICHOLS, D. A., SATYANARAYANAN, M., SIDEBOTHAM, R. N., AND WEST, M. J. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 51–81.

[15] HOWELLS, D. FS-Cache: a network filesystem caching facility. In *Linux Symposium* (Ottawa, Canada, July 2006).

[16] http://www.fusionio.com/products/ioturbine-virtual.

[17] JOSEPHSON, W. K., BONGO, L. A., FLYNN, D., AND LI, K. DFS: a file system for virtualized flash storage. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2010), pp. 85–100.

[18] KISTLER, J. J., AND SATYANARAYANAN, M. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems 10*, 1 (Feb. 1992), 3–25.

[19] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2013), pp. 45–58.

[20] LE, D., HUANG, H., AND WANG, H. Understanding performance implications of nested file systems in a virtualized environment. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 87–100.

[21] LEE, E., BAHN, H., AND NOH, S. H. A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory. *ACM Transactions on Storage 10*, 1 (Jan. 2014), 1:1–1:17.

[22] LU, L., ZHANG, Y., DO, T., AL-KISWANY, S., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Physical disentanglement in a container-based file system. In *USENIX Symp. on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014).

[23] MANN, T., BIRRELL, A., HISGEN, A., JERIAN, C., AND SWART, G. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems 12*, 2 (May 1994), 123–164.

[24] MEYER, D. T., AGGARWAL, G., CULLY, B., LEFEBVRE, G., FEELEY, M. J., HUTCHINSON, N. C., AND WARFIELD, A. Parallax: virtual disks for virtual machines. In *ACM European Conf. on Computer Systems* (Glasgow, Scotland, UK, Apr. 2008), pp. 41–54.

[25] MEYER, D. T., WIRES, J., HUTCHINSON, N. C., AND WARFIELD, A. Namespace Management in Virtual Desktops. *USENIX; login: 36*, 1 (Feb. 2011), 6–11.

[26] MICKENS, J., NIGHTINGALE, E. B., ELSON, J., NAREDDY, K., GEHRING, D., FAN, B., KADAV, A., CHIDAMBARAM, V., AND KHAN, O. Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications. In *USENIX Symp. on Networked Systems Design and Implementation* (Seattle, WA, Apr. 2014), pp. 257–273.

[27] NELSON, M. N., WELCH, B. B., AND OUSTERHOUT, J. K. Caching in the Sprite network filesystem. *ACM Transactions on Computer Systems 6*, 1 (Feb. 1988), 134–154.

[28] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *USENIX Symp. on Internet Technologies and Systems* (Seattle, WA, Mar. 2003), pp. 1–15.

[29] PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D., AND HITZ, D. NFS version 3 design and implementation. In *USENIX Summer Technical Conference* (Boston, MA, June 1994), pp. 137–152.

[30] PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *USENIX Symp. on Networked Systems Design and Implementation* (San Jose, CA, May 2006), pp. 353–366.

[31] QIN, D., BROWN, A. D., AND GOEL, A. Reliable writeback for client-side flash caches. In *USENIX Annual Technical Conf.* (Philadelphia, PA, June 2014), pp. 451–462.

[32] RAJIMWALE, A., CHIDAMBARAM, V., RAMAMURTHI, D., ARPACI-DUSSEAU, A., AND ARPACI-DUSSEAU, R. Coerced cache eviction and discreet-mode journaling: Dealing with misbehaving disks. In *Intl. Conf. on Dependable Systems and Networks* (Hong Kong, China, June 2011), pp. 518–529.

[33] SATANARAYANAN, M. Scalable, secure, and highly available distributed file access. *Computer 23*, 5 (May 1990), 9–21.

[34] SHAMMA, M., MEYER, D. T., WIRES, J., IVANOVA, M., HUTCHINSON, N. C., AND WARFIELD, A. Capo: Recapitulating storage for virtual desktops. In *USENIX Conf. on File and Storage Technologies* (San Jose, CA, Feb. 2011), pp. 31–45.

[35] TARASOV, V., HILDEBRAND, D., KUENNING, G., AND ZADOK, E. Virtual machine workloads: The case for new benchmarks for NAS. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2013), pp. 307–320.

[36] TARASOV, V., JAIN, D., HILDEBRAND, D., TEWARI, R., KUENNING, G., AND ZADOK, E. Improving I/O performance using virtual disk introspection. In *USENIX Workshop on Hot Topics in Storage and File Systems* (San Jose, CA, June 2013).

[37] THE AUSTIN GROUP. *POSIX.1-2008 Volume 2: System Interfaces*. IEEE Std 1003.1 and The Open Group Base Specifications Issue 7, 2008.

[38] VAGHANI, S. B. Virtual machine file system. *ACM SIGOPS Operating Systems Review 44*, 4 (Dec. 2010), 57–70.

[39] VAN MOOLENBROEK, D. C., APPUSWAMY, R., AND TANENBAUM, A. S. Towards a flexible, lightweight virtualization alternative. In *ACM Intl. Systems and Storage Conf.* (June 2014), pp. 8:1–8:7.

[40] VRABLE, M., SAVAGE, S., AND VOELKER, G. M. Bluesky: A cloud-backed file system for the enterprise. In *USENIX Conf. File and Storage Technologies* (San Jose, CA, Feb. 2012), pp. 237–250.

[41] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *USENIX Symp. on Operating Systems Design and Implementation* (Seattle, WA, Nov. 2006), pp. 307–320.

[42] ZHENG, W., TU, S., KOHLER, E., AND LISKOV, B. Fast databases with fast durability and recovery through multicore parallelism. In *USENIX Symp. on Operating Systems Design and Implementation* (Broomfield, CO, Oct. 2014), pp. 465–477.