File-Level, Host-Side Flash Caching with Loris

Raja Appuswamy, David C. van Moolenbroek, Sharan Santhanam, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam, Netherlands
{raja, dcvmoole, s.santhanam, ast}@cs.vu.nl

***Abstract*—As enterprises shift from using direct-attached storage to network-based storage for housing primary data, flash-based, host-side caching has gained momentum as the primary latency reduction technique. In this paper, we make the case for integration of flash caching algorithms at the file level, as opposed to the conventional block-level integration. In doing so, we will show how our extensions to Loris, a reliable, file-oriented storage stack, transform it into a framework for designing layout-independent, file-level caching systems. Using our Loris prototype, we demonstrate the effectiveness of Loris-based, file-level flash caching systems over their block-level counterparts, and investigate the effect of various write and allocation policies on the overall performance.**

***Index Terms*—Caching, Flash, File System Architecture, SSD, NAS, Loris**

I. Introduction

Over the past few years, many enterprises have shifted from using direct-attached storage to network-based storage for housing primary data. By providing shared access to a large volume of data and by consolidating all storage resources at a single spot, network-based storage improves scalability and availability significantly. The storage industry has also witnessed an equally phenomenal increase in the adoption of flash-based solid state storage. While flash can be used in several capacities (data/metadata caches, primary storage devices, etcetera) in a networked storage server, recent research has shown that using flash at the host rather than server side has several advantages [4], [14]. First, a hit on the host-side flash cache can be serviced immediately without an expensive network access. Second, by filtering requests, a host-side cache significantly reduces the number of requests that need to be serviced by the server. By eliminating bursty traffic, host-side caching enables servers to be provisioned for average I/O volumes, rather than peak volumes, thereby reducing capital expenses.

Figure 1 shows the architecture and components involved in a typical host-side caching implementation. As shown in the figure, storage resources consolidated at the server side are exported to the host side using a Storage Area Network (SAN) protocol like iSCSI. File systems at the host, which traditionally managed direct-attached storage devices, are now used to manage remote storage volumes. Several systems integrate caching into the storage stack below the file system and above the iSCSI client, thereby retaining backward compatibility with existing file systems.

Block-level caching systems can be classified into two types depending on their write policy, namely, *write through* and *write back*. A write-through cache issues writes to both the networked primary storage and the local flash cache in the order in which they were received and waits for these writes to complete before passing back an acknowledgement to the file system. A write-back cache, on the other hand, acknowledges writes as complete as soon as they are serviced by the local flash cache. The networked primary storage is updated asynchronously in background. Under write-intensive workloads, write-back caching is guaranteed to improve performance as it converts high-latency, foreground writes into asynchronous, low-latency background writes. However, as a block-level, write-back cache intercepts and caches file system requests, it absorbs writes to both metadata and data blocks alike, thereby causing several problems as it silently changes the file system-enforced data ordering.

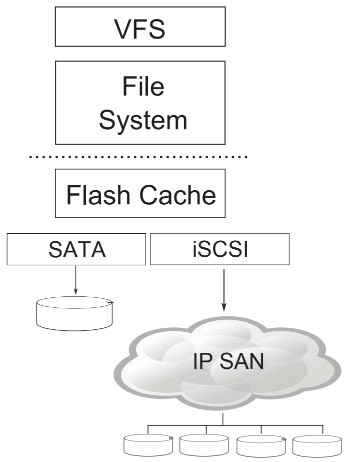


Figure 1: Traditional host-side flash caching architecture. The figure shows pooled storage resources at the server side exported to the host side over an IP-based Storage Area Network. On the host side, the dotted lines demarcate file-aware layers from those that are not. Thus, the flash cache is managed by a file-unaware, block-level cache driver that uses the SATA and iSCSI subsystems to communicate with the local cache device and remote primary storage. The semantically-aware file system remains oblivious to the usage of a cache device.

The first issue is the impact on the correctness of several server-side administrative operations such as backup, snapshoting and cloning. For instance, a snapshot initiated at an inopportune time might capture an inconsistent image of the data volume. Thus, backups made off this snapshot would not help in disaster recovery, as the host file system might not be able to fix the inconsistency in the restored snapshot. Second, a loss of file system metadata due to an SSD failure could have a negative impact on availability as it can cause substantial data loss. Third, almost all state-of-the-art enterprise storage systems adopt the "release consistency" model [4] when multiple storage clients access the same storage volume. Under this model, a volume is shared across clients in a serial fashion with only one client maintaining exclusive access at any time. When block-level write-back caching is used in such a setting, the failure of one client can render the network storage inconsistent or, in the worst case, unusable by any other client. In light of these issues, it is not surprising that several write-back caching implementations even make explicit disclaimers warning administrators about storage-level inconsistencies after a cache failure [6].

A. Consistent, Block-Level Write-Back Caching

To solve these issues caused by unordered write-back policy, two solutions were proposed recently [12]. The first technique, referred to as *Ordered Write Back*, is based on the simple idea that consistency at the networked storage system can be maintained by evicting blocks in the same order in which they were received by the cache. The intuition behind this idea is the fact that file systems already maintain a consistent disk image by enforcing ordering of write requests (for example, journaled writes must precede actual data writes). However, this approach has several bottlenecks that impose a significant performance penalty. For instance, the cache must keep track of dependencies between data blocks – an operation with non-trivial compute and memory requirements. It must also preserve and write back all dirty copies of the same block, thereby wasting cache space and network bandwidth.

To solve problems with Ordered Write Back, *Journaled Write Back* has been proposed. The idea behind this approach is to use a host-side, persistent journal, in concert with a journal on the server side, to bundle file system updates into transactions that are checkpointed asynchronously to remote storage in the background. Thus, the Journaled Write Back approach enables consistent, block-level, write-back caching at the host side only when used in concert with networked servers that provide a atomic-group-write interface. In addition, under certain configurations, this approach would suffer from performance issues due to redundant use of journaling by both the file system and block-level cache to protect the same data and metadata blocks. Recently, researchers have shown how such redundant journaling, albeit in a different context (SQLite database and ext4 file system), deteriorates application performance significantly in the Android stack [9].

B. Filesystem-Based Caching

Given that all modern file systems use techniques like journaling and shadow copying to ensure metadata consistency across reboots, the natural alternative to integrating caching at the block level is to modify existing file systems to be cache aware. However, such an integration has one major issue – its lack of portability. A caching algorithm integrated into a file system is restricted to work only within the scope of that file system. This lack of portability would only be an inconvenience rather than a show stopper if device heterogeneity were nonexistent.

Heterogeneity exists both within and across device families. New devices, with interfaces different from the traditional block-based read/write interface, are emerging in the storage market. For instance, some flash devices and Storage Class Memory devices are byte accessible, while Object-based storage devices, on the other hand, work with objects rather than blocks. Integrating these devices into the storage stack requires building custom file systems, and hence reimplementing the caching algorithm, for each device family.

Similarly, different SSDs, sometimes even from the same vendor, have different performance characteristics. For instance, Intel X25-V SSD design makes a price/performance trade-off, as it sacrifices sequential read/write throughput by reducing the number of channels populated with NAND flash. Intel X25-M, on the other hand, has equally impressive random and sequential read/write performance figures. Achieving optimal performance in such cases requires pairing devices with their ideal layout algorithms. For instance, a log-structured layout might be best suited for an Intel X25-M, while it could deteriorate performance when used with X25-V. This heterogeneity in layout management forces one to reimplement caching algorithms not just across device families, but also for each new layout algorithm within device families.

C. Our Contributions

Solving the heterogeneity issues faced by file system-based caching solutions requires decoupling flash-cache management from layout management. In prior work, we proposed Loris [2], a fresh redesign of the storage stack that implements layout-independent, file-level RAID algorithms. In this paper, we present our design extensions to Loris that transform it into a framework for implementing layout-independent caching solutions. In doing so, we make three major contributions to state of the art.

First, in contrast to traditional approaches, we make the case for integrating caching algorithms at a different level in the storage stack (Section II). With the new integration, caching algorithms work at a higher level of abstraction by managing files rather than disk blocks. As we will see later in this paper, one of the major challenges with such an integration is implementing efficient subfile caching. Thus, our second contribution is the Loris-based subfile caching framework that can be used by any caching algorithm to map each logical file block to a different storage target (Section III). Our third contribution is a thorough comparative evaluation of our Loris prototype with a block-level solution to prove the effectiveness of our approach against a traditional block-level cache, and to understand the impact of caching policies on overall performance (Section IV).

File-level caching has been implemented earlier in the context of distributed file systems like AFS [8] and Coda [11]. In these systems, the client implementation runs as a user-space application and uses the local file system to perform coarse-grained, whole-file caching of application data (not system metadata) stored in a networked file store. We, on the other hand, integrate flash-caching algorithms directly into

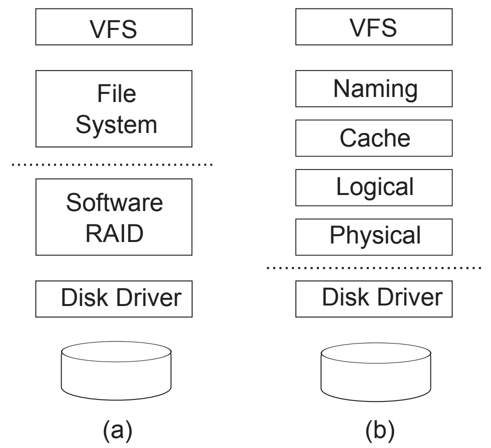


Figure 2: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

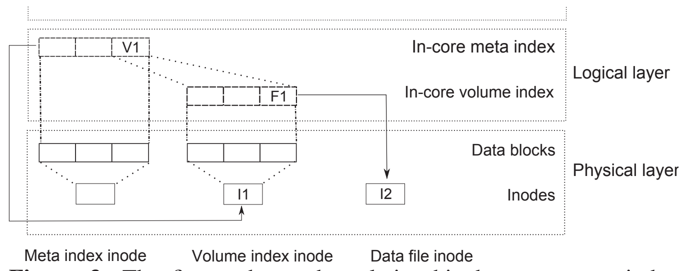


Figure 3: The figure shows the relationship between meta index and volume index. The figure shows the meta index file containing the file volume metadata entry for volume V1, which could be <V1, REGULARVOL, volume index configuration=<raidlevel=1, stripesize=N/A, physicalfiles=<D1:I1>>. Thus, inode I1 in physical module D1 is used to store the volume index file data (an array of logical file configuration entries) for file volume V1. The logical file configuration entry for file <V1, F1> could be <raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I2>>. Thus, inode I2 in physical module D1 is used to store file F1's data.

the local storage stack and show how a file-level (but not whole file) integration of caching algorithms can be used to implement unified, block-granular caching of both application data and system metadata, without any of the consistency issues or performance overheads of the traditional block-level integration.

II. The Case For File-level Host-Side Caching With Loris

In this section, we will first provide a quick overview of the Loris storage stack. Following this, we will show how Loris makes layout-independent integration of flash caching possible, and we will make the case for such an integration by describing its advantages over file system-based and block-level approaches.

A. Loris - Background

Loris is made up of four layers as shown in Figure 2. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Every Loris file is uniquely identified using a <volume identifier, file identifier> pair. Each Loris file belongs to a file volume, which is a rooted collection of files and directories. Each Loris file is also associated with several *attributes*, and the interface supports two attribute manipulation operations—*getattribute* and *setattribute*. Attributes enable information sharing between layers, and are also used to store out-of-band file metadata. We will now briefly outline the responsibilities of each layer in a bottom-up fashion.

1) Physical Layer: The physical layer exports a physical file abstraction to the logical layer. A physical file is a stream of bytes that can be read or written at any random offset. Thus, details such as the device interface and on-disk layout are abstracted away by the physical layer.

Each physical layer implementation is tasked with providing 1) device-specific layout schemes for persistent storage of files data/attributes, and 2) end-to-end data verification using parental checksumming. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*. Our current physical layer prototype is based on the traditional UNIX file system layout. Each physical file is represented by an inode. Each inode contains enough space to store the file's Loris attributes, seven direct data block pointers, and one single, double and triple indirect block pointer. Free blocks/inodes are tracked using block/inode bitmaps. Although our physical layer implements parental checksumming of all data and metadata, we will omit the details as we do not use it in our evaluation.

2) Logical Layer: The logical layer exports a *logical file* abstraction to the cache layer. A logical file is a virtualized file that appears to be a single, flat file to the cache layer. Details such as the physical files that constitute a logical file, the RAID levels used, etc. are confined within the logical layer. The logical layer works with physical files to provide both device and file management functionalities. It is made up of two sublayers, namely the file pool sublayer at the bottom, and the volume management sublayer at the top.

In prior work, we introduced a new Loris-based storage model called File Pooling, that simplifies management of storage devices [3]. File pools simplify storage administration and enable thin provisioning of file volumes [3]. The file pool sublayer maintains data structures necessary for tracking device memberships in file pools, and provides device management operations for online addition, removal and hot swapping of devices.

The volume management sublayer supports file volume virtualization. As we mentioned earlier, each logical file belongs to a file volume. Each file volume is physically represented by a *volume index file* which is created at file volume creation time. This file stores *file configuration information* entries for all files belonging to its volume. This configuration information consists of 1) RAID level used, 2) stripe size used (for certain RAID levels), and 3) list of physical files that store the logical file's data.

Similar to the way the volume index file tracks the membership of files in file volumes, file volumes themselves are tracked by the *meta index file*. This file contains *file volume* *metadata* entries, one per volume, that record: 1) the number of files in that volume, 2) tiering/caching policy used, and 3) physical file(s) that store the volume index data among other details. Figure 3 describes the relationship between these two data structures with an example.

3) Cache and Naming Layers: The cache layer provides in-core caching of data pages. Our prototype cache layer implements the LRU cache replacement algorithm.

The naming layer acts as the interface layer. Our prototype naming layer implements the traditional POSIX interface by translating POSIX files and attributes into their Loris counterparts. It implements the directory abstraction by using Loris files to store directory entries. All POSIX semantics are confined to the naming layer. However, the naming layer uses the attribute infrastructure to help the other layers discern Loris metadata from application data. For instance, as far as the logical layer is concerned, directories are just regular files with special attributes that mark them as important. It uses this information to mirror directories on all physical layers for improving availability.

4) Crash Recovery in Loris: Similar to other systems, Loris also uses snapshot-based recovery to maintain metadata consistency across system failures. Due to lack of space, we will just present an overview here and we would like to direct the reader to [17] for further details. During normal operation, after every preconfigured time interval, Loris takes a system-wide snapshot. During this operation, all Loris layers flush out any dirty data and metadata that is yet to be written. Then, the logical layer asks each physical module to take a snapshot of all metadata (both physical module's layout-specific metadata and those belonging to the other Loris layers) and tag the snapshot with a common timestamp. It is important to note here that only metadata, not application data, is snapshotted and the physical layer can distinguish metadata from data using attributes as we mentioned earlier. Thus, each global metadata snapshot can be identified using a single timestamp across all physical modules.

After a system failure, logical layer probes all physical modules for their latest timestamp. If the system had shutdown gracefully, all physical modules would return back the same timestamp. A disparity in timestamp indicates an unclean shutdown, upon which the logical layer instructs all physical modules to roll back metadata to the latest common timestamp. Thus, in a nutshell, the task of providing consistency in Loris is divided between the logical and physical layers. Each physical module is tasked with implementing some form of metadata snapshoting. The logical layer works with physical module snapshots and coordinates recovery to a globally consistent snapshot after a system failure.

B. File-level Host-side Caching With Loris

Comparing Loris with the traditional stack (Figures 1, 2), one can observe two things. First, the file system, which is a monolithic module in the traditional stack, has been decomposed into naming, cache, and physical layers in the Loris stack. Second, as the dotted line indicates, all Loris

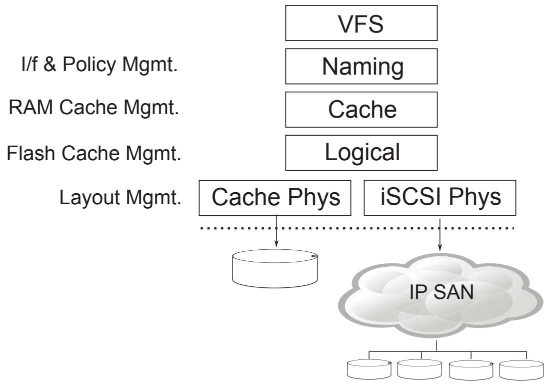


Figure 4: Host-side flash caching with the Loris stack. The figure shows the roles and responsibilities of each layer when the Loris stack is used as a host-side caching solution. Contrasting this with Figure 1, one can see that the local flash cache is managed by the file-aware logical layer.

layers operate at the file level in contrast to the traditional stack, where RAID and caching algorithms operate at the block level. It is because of these two fundamental design differences that Loris enables a new level (the logical layer) at which flash caching can be integrated.

Figure 4 shows how Loris can be used as a host-side cache. Loris runs on the host machine as the primary file system and manages both the local SSD and the remote iSCSI volume. Physical layer implementations customized for SSD and iSCSI storage map device blocks to Loris physical files. The caching logic (allocation and replacement algorithms), however, is implemented at the file level, in the logical layer, in contrast to the traditional caching design where it is integrated at the block level. This integration possesses all the advantages of a file system-based approach without any of its disadvantages.

First, the file-level implementation of caching makes it device or storage interface agnostic. Switching to a new type of caching device (like MEMS or Object-based Storage (OSD) instead of SSD) requires just implementing corresponding physical modules. Thus, file-level caching obviates translation layers as there is no necessity to map any device interface to a generic block interface. In the absence of such abstractions, device-specific physical layer implementations can implement highly-customized optimizations that exploit advantages specific to each device family. For instance, one could implement a short-circuit-shadow-paging-based physical layer for a PCM device [5], or a physical layer that exploits the virtualized-flash-storage abstraction offered by modern PCI Express SSDs [10], without affecting the caching implementation. Later in this paper, to show the benefit of interface-agnostic flash caching, we will describe our implementation of a simplified NFS-client-like physical layer that enables to usage of any networked, file-based remote storage system as primary data store.

During normal operation, the logical layer treats all physical modules (local and remote) alike and establishes global metadata checkpoints across them. Irrespective of where they are stored, all metadata updated between two checkpoints get persisted as a part of the next global checkpoint, or reverted during recovery after a crash, as a single atomic unit. Thus, the second benefit is that any Loris-based caching implementation can recover from OS crashes and power failures on the host side using Loris' built-in consistency mechanism without any additional effort.

We would like to explicitly mention here that these consistency-enforcing metadata checkpoints are created and maintained by the physical layer implementations and thus, have no influence on logical layer-resident, flash-cache management algorithms that perform caching of application data. Also, these checkpoints are different from administrator-triggered file volume snapshoting of all data and metadata. While Loris is capable of supporting such snapshoting, and while the interaction between snapshoting and caching certainly requires special attention [4], our focus in this paper is on using Loris as a host-side cache with any server-side NAS or SAN appliance. In this scenario, administrators typically use server-side (not client-side) snapshoting facilities for performing various administrative operations. We intend to integrate host-side snapshoting with the caching framework described in this paper as a part of future research which involves investigating the utility of Loris as a hypervisor flash cache in virtualized data center (Section V).

Third, by being file aware, Loris can use different caching policies for different file types. For instance, Loris could associate all metadata with the write-through policy. Thus, even if the user specifies a write-back policy for all application data, writes by the naming layer to directory files and by the logical layer to volume index file will be written through immediately to the networked storage server. By having all metadata written through immediately, Loris can recover from all host side failures. For instance, a failure of the SSD on the host side would only result in application data loss and never renders the networked primary storage unusable. Inconsistencies between data and metadata caused by an SSD failure can be easily identified by the logical layer and propagated to the application on demand, thereby providing high availability. Thus, Loris provides a framework for implementing persistent, file-level write-back caching systems that do not suffer from any of consistency issues that plague the block-level integration.

Fourth, as the caching algorithms are plugin-based, Loris can easily pair workloads with ideal caching algorithms in contrast to even modern, state-of-the-art cache-aware file systems that adopt a single, one-size-fits-all approach to flash caching. This flexibility is especially important in modern data centers where server consolidation forces a single host-side caching system to service requests from disparate workloads with different RPOs. For instance, ZFS [1] uses SSDs that can sustain high random IOPS for caching read-only data and SSDs with high sequential write throughput for storing the ZFS journal (ZIL) [13] irrespective of application workload. Loris, on the other hand, could pair workloads with write-back or write-through caching depending on their RPO.

Although Loris provides a convenient framework for implementing host-side caching systems, the whole-file nature of mapping maintained by the logical layer makes it impossible

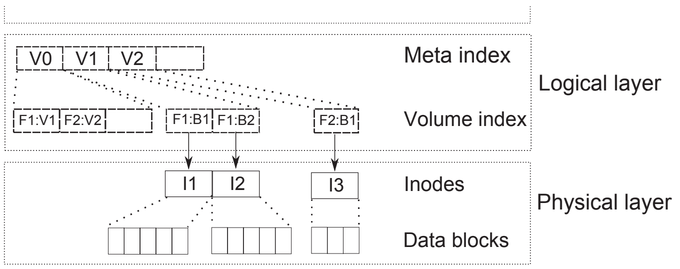


Figure 5: File-as-a-volume subfile mapping approach: The figure shows blocks F1:B1, F1:B2 of file $<$ V0, F1$>$, and block F2:B1 of file $<$ V0, F2$>$ are mapped to physical files I1, I2 (for F1), and I3 (for F2).

to implement fine-grained, subfile caching, as caching a single file block requires caching the entire file. Even worse if the fact that such subfile caching is mandatory for implementing write-back caching, where files can be arbitrarily written/updated in small chunks. Thus, the logical layer must be completely redesigned to support fine-grained, subfile caching.

III. Loris-based Host-side Cache: Architecture

Our new logical layer consists of three plugin-based sublayers: 1) the volume management sublayer with extended support for subfile caching, 2) the cache management sublayer that manages the flash-based host-side cache, and 3) the file pool sublayer which provides device management and RAID services. Each of these sublayers has a well-defined interface, similar to the Loris interface, and can be replaced without changing the other sublayers. As device management and RAID algorithms are out of the scope of this paper, we will now describe in detail the first two sublayers.

A. Volume Management Sublayer: Subfile Mapping

As we mentioned earlier, our original logical layer maps each logical file to one or more physical files. However, implementing subfile caching requires mapping each logical file block (not the whole file) to one or more physical files. In addition, we also need to maintain metadata that identifies the block cached in the SSD as clean or dirty; the action taken during background synchronization and cache eviction varies depending on the block state (clean data can be just discarded whereas dirty data must be written back to primary storage).

As the relationship between files and blocks is very similar to the relationship between file volumes and files, we initially implemented this indirection by recursively extending the file volume abstraction. When a logical file was created, an entry was allocated for it in its parent file volume's volume index as before. In addition, we created a new volume, whose logical configuration information entries record the logical block–physical file(s) mapping for each block as shown in Figure 5. Thus, by treating each logical file as a file volume, we were able to extend the existing abstraction to support subfile mapping with minimal effort.

However, preliminary evaluation revealed that the overhead caused by metadata allocation and lookup was a significant source of performance degradation. With the aforementioned subfile mapping, we were storing physical file information for each logical block. Each physical file is represented by a <moduleid, inode number> pair, which is encoded using four bytes in our current implementation. Assuming an block size of 4-KB, and assuming that all blocks are cached on the SSD, a 4-GB file would require 8-MB (eight bytes per block, four for HDD physical file and four for the SSD one).

To eliminate the metadata overhead, we increased the mapping granularity from a block to an extent - a logically contiguous group of blocks. While this did improve performance significantly, it still suffered from two problems. First, as each extent was stored as a separate physical file, at small extent sizes, most read/write requests would need to be serviced by reading/writing multiple physical files. Despite the fact that our prototype exploits the Native Command Queuing (NCQ) capability of both HDD and SSD by queuing these reads/writes in parallel, we found that splitting a single, large read/write request into several constituent extent-sized requests had a significant performance impact. Second, extent-granular mapping complicates write-back caching as caching algorithms and staging/eviction of data must be extent aligned. In addition, writes misses force the entire extent to be read from the disk, if not already cached, causing performance deterioration at large extent sizes. Thus, we abandoned the "file-as-a-file-volume" approach and adopted a newer one.

Our new approach is based on the insight that rather than storing each logical block in a separate physical file, we could use a single physical file to pack related blocks. In other words, each logical file could be associated with two physical files, one on the SSD, and the other on the iSCSI volume. By doing so, for each logical block, we would need to record whether 1) it has been cached on the SSD physical file, and 2) if the SSD copy is dirty. We could easily accomplish this with just two bits per block. Thus, in contrast to the previous approach, a 4-GB file can be encoded using 128-KB for caching status bits, 128-KB for the dirty bits, and 8 bytes for the physical file information. Thus, we reduce the metadata footprint by a factor of 32 (8 bytes to 2 bits per entry). In addition, a large request spanning multiple blocks would now translate to a single physical file read (assuming that the corresponding blocks are all sequential and collocated in the same device).

In our current prototype, we implemented this new mapping by modifying the file configuration information stored by the volume management sublayer. The new configuration information contains 1) physical file information for the primary file in the iSCSI physical layer, 2) physical file information for the cached SSD file (if cached), 3) a 32-byte block status bitmap, and 4) a 32-byte dirty bitmap. In order to implement fine-grained caching, we use a block size equal to the OS page size of 4-KB. Thus, the block information for all files up to 1-MB resides entirely within its logical configuration information. When a file files grows over 1-MB, we dynamically create a new physical file and use it to store both bitmap blocks.

B. Cache Management Sublayer

There are two main design parameters that influence the operation of a host-side caching system. The first parameter is the write policy (write through or write back, as we mentioned in Section I). The second parameter is the allocation policy that controls when data is admitted into the cache. Based on this policy, caches can be classified as *write-allocate* and *write-no-allocate* (also known as *write-around*). As the names imply, the former policy admits data on write misses while the latter does not. As we wanted to systematically study the effect of each parameter on overall performance, we implemented all four possible host-side caching alternatives as a separate cache management plugin.

Although these plugins differ with respect to write and allocation policies, they all share two things in common. First, they all admit data into the cache as a side effect of a read miss. Second, they all use the LRU replacement algorithm to manage the flash cache. In our current prototype, the algorithm is implemented as a separate component independent of all plugins. It maintains an in-memory list of entries, on per logical file block, in LRU order. As this list does not contain information about the physical location of logical file blocks (which is recorded in volume index entries and protected using Loris' consistency mechanism), it can be maintained entirely in memory as a power failure or system crash would result only in the loss of recency information. In addition, as each list entry only needs to store the logical file id and offset for currently cached blocks, it can easily scale to large cache sizes.

C. Physical Layer Support For Subfile Caching

The aforementioned changes to the logical layer make fine-grained cache admission possible. However, as caching algorithms work at a page granularity, they also require the capability to evict individual pages for freeing up cache space. As caching algorithms operate at the logical layer, there were two ways a Loris-based cache implementation could free up space, namely, deleting a physical file or truncating it. We found both these operations to be too coarse grained and inefficient as it is impossible to implement fine-grained individual page/block evictions using either of these methods.

To solve these problems, we added a new *rdelete*(range delete) operation to the physical layer API. Only those physical modules that will be used to manage cache devices need to support this operation. The logical layer uses rdelete to free arbitrary data ranges in physical files. When space is needed in the flash cache for accommodating new data, the LRU algorithm is invoked to find the longest sequence of logically contiguous file blocks (blocks belonging to the same logical file at consecutive offsets). The logical layer then issues an rdelete call to the physical module that manages the flash cache which, in turn, frees those data blocks and associated indirects, effectively creating holes in that physical file.

D. Network File Store

Although our traditional Loris physical layer can be used over an iSCSI volume, we wanted to prove the utility of layout-independent, file-level flash caching. So, we implemented a new physical module that can communicate with any file server that support four basic calls to create, delete, read from and write to a file. This network file client is essentially a simplified NFS client that maps calls from the Loris interface to the limited file server interface. To make our client portable across file servers, we also implemented support for attribute handling at the client side. Each client maintains a special file, which is created during startup, in which it stores the attributes for all Loris files. Thus, while read/write calls for Loris files get mapped onto file server read/write calls, getattr and setattr operations directed at the network client are implemented reading/writing into this special file.

IV. Evaluation

In this section, we will first describe the hardware/software setup and benchmarking tools following which we will present our comparative analysis of the four Loris-based cache management architectures to understand the impact of each policy on overall performance.

A. Setup

The client machine we used in all tests was an Intel Core 2 Duo E8600 PC, with 4-GB RAM. We used a OCZ Vertex3 Max IOPS SSD as our host-side flash cache. Our networked file server is an Intel Core2Duo E8600 PC, with 4-GB of RAM and a 500-GB 7200 RPM Western Digital Caviar Blue (WD5000AAKS) SATA drive. In all experiments, we used the first 8-GB of both SSD and HDD to house all data.

Both client and server machines run Minix 3, a microkernel, multiserver operating system [15]. Loris runs on the client machine and manages the flash cache while we use the Minix 3 File System on the server machine as our network file store. We deliberately configured Loris (on the client side) to run with only 64-MB RAM cache (cache layer) to ensure that our flash-caching subsystem (in Logical and Physical layers) is stressed thoroughly. As we vary the flash cache size in some experiments, we will report the actual SSD size used while describing the results of each experiment.

Although we implemented all the features we described in Section III in our prototype, we will focus only on the performance aspect of Loris-based file-level caching in this paper.

B. Benchmarks and Workload Generators

We used Postmark and FileBench to generate four different classes of server workloads for our comparative evaluation. Postmark is a widely used, configurable file system benchmark that simulates a mail server workload. We configured Postmark to perform 80,000 transactions on 40,000 files, forming a dataset of roughly 1-GB, spread over 10 subdirectories, with file sizes ranging from 4-KB to 28-KB, and read/write granularities of 4-KB. We report the transaction time, which excludes the initial file preallocation phase, for all cases.

FileBench is a flexible, application-level workload simulator. We used two predefined workload models to generate File Server and Web Server workloads. For the File Server workload, we configured FileBench to generate 10,000 files, using a mean directory width of 20 files, and a median file size of 128-KB. The workload generator performs a sequence of create, write, read, append (using a fixed I/O size of 1-MB), delete and stat operations, resulting in a write-biased workload. The Web Server configuration generates 25,000 files, using a mean directory width of 20 files. The median file size used is 32-KB, which results in a workload dominated by small file accesses, with the exception being an append-only log file. The workload generator performs a sequence of ten whole-file read operations, simulating reading web pages, followed by an append operation (with an I/O size of 16-KB) to a single log file. Unlike Postmark, the total data size generated by the FileBench workloads varies between 1.5-GB and 4-GB depending on the performance of the underlying caching system.

C. Comparative Evaluation: Caching Policies

Our first goal in evaluating the Loris prototype is to understand the impact of various caching policies on overall performance. Table I shows the IOPS/execution time achieved by various benchmarks under Loris in the networked configuration. These results were obtained by fixing the host-side flash cache to 1.5-GB. There are three important observations to be made from Table I.

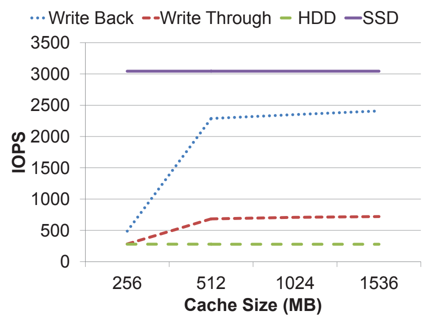
Benchmark	WB	WB-WA	WT	WT-WA
Postmark (secs)	458	2084	1348	2304
File Server (IOPS)	1046	231	255	199
Web Server (IOPS)	3423	2856	3472	2859

Table I: Execution time (in seconds) and IOPS achieved by write-back (WB) and write-through (WT) cache plugins, with and without write-around (WA) allocation, under various benchmarks, at a cache size of 1536-MB.

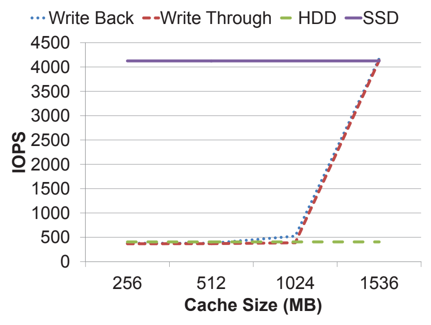
First, notice that both write-back and write-through caching offer identical performance under Web Server. This is because of the fact that at 1.5-GB, the flash cache has a 100% read hit rate and all benchmark reads are satisfied entirely on the host side under both write-back and write-through schemes. The only writes under Web Server are appends to the log file which are never read back. Thus, this result proves that our caching algorithms work as expected.

Second, clearly, write-back caching has a significant impact on overall performance. It registers an impressive 310% improvement in IOPS under File Server and 66% reduction in execution time under Postmark. This clearly indicates the importance of using the host-side flash cache as a read-write cache (as opposed to a read-only cache).

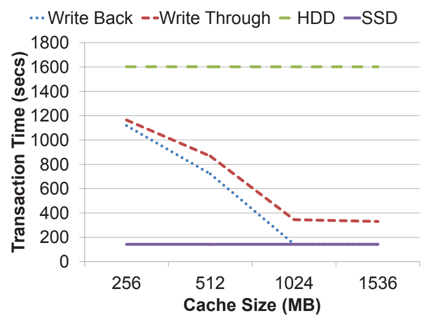
Third, notice that write-around allocation policy consistently deteriorates performance of both write-back and write-through caching under all benchmarks. Without the write-around policy, all write misses allocate data in the SSD under both write-back and write-through caching policies. Thus, when the cache is large enough to hold a substantial portion of the working set, most read/write requests can be satisfied entirely on the host side. But with write around enabled, cache allocation happens only as a side effect of a read miss. Thus, the first read request



(a)



(b)



(c)

Figure 6: Figures show the IOPS/transaction time achieved by Loris under both single-device (HDD/SSD) and multi-device caching configurations at various cache sizes in MBs (axis labels) under the File Server (a), Web Server (b) and PostMark (c) benchmarks.

for each data block has to be serviced by the networked file server, thereby reducing performance. As we found this to be the general case irrespective of the benchmark/experimental setting, we will not discuss write-around caching any further.

D. Network Performance Sensitivity

One of the rationales used by proponents of write-through caching is that the performance advantage of write-back caching would only be marginal, at best, when deployed in a setting with high-speed network interconnects and storage backends. In order to understand the sensitivity of write-caching policies with respect to network performance, we reran the experiments on the host machine using a direct-attached SATA HDD instead of the network file server as the storage backend. In this configuration, we used the default Loris physical module we described earlier to manage the layout of both SSD and HDD.

Looking at the results under cache size 1536-MB in Figure 6, one can see that write-back caching still produces a 56% reduction in execution time under Postmark and a 234% increase in IOPS under the File Server benchmark. Based on these results, we believe that write-back caching is valuable even when used in a setting with low-latency interconnects and high-performance storage backends.

E. Cache Size Sensitivity

While write-back caching would have a clear edge over its write-through counterpart at cache sizes where the read hit rate is 100%, it is important to understand if it is still beneficial at smaller cache sizes. In order to do so, we reran the experiments in the direct-attached HDD configuration while varying the flash cache size. Figure 6 reports the IOPS/execution time of write-back/write-through caching policies under various benchmark-cache size combinations.

There are three important observations. First, one can see that even at the smallest cache size, write-back caching produces a 30% reduction in execution time under PostMark and a 75% increase in IOPS under File Server when compared to the disk-only case, thereby proving the utility of caching.

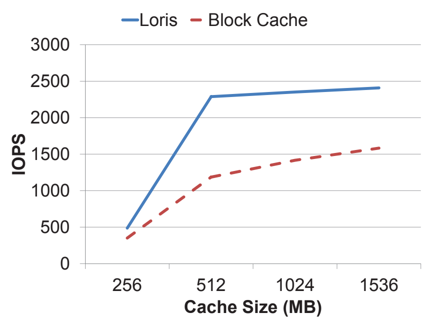
Second, as exemplified by the Web Server benchmark, read-intensive benchmarks derive no benefit from write-back caching. This is expected as the only writes under this benchmark are those issued to the append-only log that is never read. Thus, one might as well resort to using write-through caching, perhaps even with existing block-level solutions, under such read-intensive benchmarks.

Third, under write-intensive benchmarks, write-back caching always matches or outperforms write-though caching, but the performance of write-back caching is extremely sensitive to both the read/write ratio and cache size. For instance, at cache size of 256-MB, while IOPS increases by a sizeable 72% under the File Server benchmark with write-back caching, Postmark results are more modest. However, at all other cache sizes, write-back caching improves execution time by 16% to 63% under Postmark.

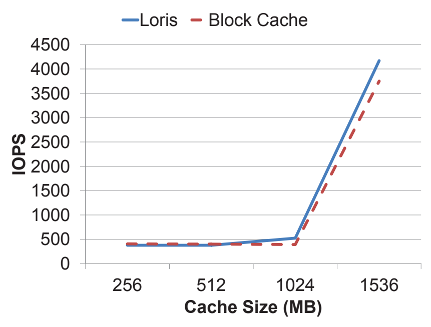
F. File-Level and Block-Level Caching Comparison

One of the goals in evaluating our Loris prototype was to prove the effectiveness of file-level caching as opposed to block-level caching. In order to do so, we implemented a LRU-based caching filter driver that is positioned between the file system and AHCI disk driver. The cache driver works similar to its file-level counterpart by maintaining an in-memory list of disk blocks in LRU order. It implements write-back caching of data stored in the direct-attached HDD by interposing file system requests and satisfying both reads and writes from the SSD if possible. To make a fair comparison, we ran Loris over the block-level cache. Thus, in this configuration, Loris is not used as a caching framework, but rather as just a regular file system that manages the logical disk exposed by the block-level cache implementation.

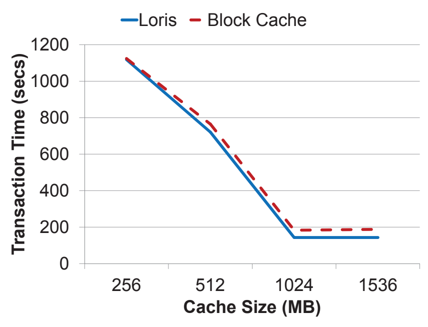
Figure 7 shows the performance of the two caching implementations. Clearly, the Loris-based file-level caching implementation outperforms its block-level counterpart by a significant margin under both Postmark at high cache sizes and under File Server at all cache sizes. On further investigation, we found out this to be due to the impact of file deletions on the two caching implementations. When a file is deleted, the Loris-based implementation frees all cached data associated with it in the SSD. In the block-level case, on the other hand, deletions are handled by the file system.



(a)



(b)



(c)

Figure 7: Figures show the IOPS/transaction time achieved by both Loris and block-level write-back caching implementations at various cache sizes in MBs (axis labels) under the File Server (a), Web Server (b) and PostMark (c) benchmarks.

Hence, the block-level cache driver never gets a notification from the file system about deleted file blocks. Thus, unlike Loris, which never migrates deleted data blocks, the block-level implementation incurs heavy penalty due to unwarranted caching and migration of useless deleted data. While this drawback could be overcome by having the file system share such semantic information with a block-level cache (using the TRIM command for instance), we would like to emphasize here that a file-level implementation has easy access to such rich semantic information out of the box. The conclusion we would like to draw from these results is that file-level caching implementations are capable of meeting the performance of their block-level counterparts without resorting to suboptimal, redundant consistency management.

V. Conclusion

As flash devices grow in density, we believe that write-back caching will become a standard feature in all future host-side caching systems. In this paper, we showed how a file-level integration of caching algorithms solves, by design, various consistency issues that plague the traditional block-level integration. Using our Loris prototype, we dispelled the myth that file-level caching implementations cannot work efficiently on a subfile basis. Although we used Loris as our framework, we would like to point out that one could theoretically build file-level caching systems using the traditional storage stack with the assistance of the stackable file system framework [7].

Given the very many benefits of file-level caching, an interesting area of future research is investigating its integration in modern virtualized data centers, where NAS-based filers are being increasingly deployed as backend stores for storing disk images of consolidated server virtual machines. Recent research has shown that unwarranted translations enforced by layers of virtualization have a huge negative impact on performance, and that such overhead could be eliminated by using a paravirtualized NAS client in the guest OS [16]. Thus, an alternative to the traditional approach of having the hypervisor perform caching of VM-disk-image blocks [4] would be to use a hypervisor-resident, file-level flash-caching implementation (like Loris) in combination with the paravirtualized NAS client to cache files rather than blocks. We intend to investigate the understand the pros and cons of such an approach as a part of future research.

References

[1] "Sun Microsystems, Solaris ZFS file storage solution. Solaris 10 Data Sheets," 2004.

[2] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Loris - A Dependable, Modular File-Based Storage Stack," in *Proc. of the 16th IEEE Pacific Rim Intl. Symp. on Dep. Comp.*, 2010.

[3] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum, "Flexible, Modular File Volume Virtualization in Loris," *Proc. of 27th IEEE Conf. on Mass Storage Sys. and Tech.*, 2011.

[4] S. Byan, J. Lentini, A. Madan, and L. Pabon, "Mercury: Host-side flash caching for the data center," in *Proc. of 28th IEEE Conf. on Mass Storage Systems and Tech.*, 2012.

[5] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *Proc. of the ACM SIGOPS 22nd Symp. on Oper. Sys. Prin.*, 2009.

[6] J. Corbett, "A bcache update, http://lwn.net/articles/497024/."

[7] J. S. Heidemann and G. J. Popek, "File-system development with stackable layers," *ACM Trans. Comp. Sys.*, vol. 12, no. 1, pp. 58–89, 1994.

[8] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West, "Scale and performance in a distributed file system," *ACM Trans. Comp. Sys.*, vol. 6, no. 1, pp. 51–81, 1988.

[9] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won, "I/O Stack Optimization for Smartphones," in *Proc. of the USENIX Ann. Tech. Conf.*, 2013.

[10] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li, "Dfs: A file system for virtualized flash storage," in *Proc. of the Eigth USENIX Conf. on File and Storage Tech.*, 2010.

[11] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Trans. Comp. Sys.*, vol. 10, no. 1, pp. 3–25, 1992.

[12] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao, "Write policies for host-side flash caches," in *Proc. of the 11th USENIX Conf. on File and Storage Tech.*, 2008.

[13] A. Leventhal, "Flash Storage Memory," *Commun. ACM*, vol. 51, 2008.

[14] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: a lightweight, consistent and durable storage cache," in *Proc. of the 17th ACM European Conf. on Comp. Sys.*, ser. EuroSys, 2012, pp. 267–280.

[15] A. S. Tanenbaum and A. S. Woodhull, *Operating Systems Design and Implementation (Third Edition)*. Prentice Hall, 2006.

[16] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok, "Virtual machine workloads: The case for new benchmarks for NAS," in *Proc. of the 11th USENIX Conf. on File and Storage Tech.*, 2013.

[17] D. van Moolenbroek, R. Appuswamy, and A. Tanenbaum, "Integrated system and process crash recovery in the loris storage stack," in *IEEE Seventh Intl. Conf. on Net., Arch. and Storage*, 2012.