Integrating Flash-based SSDs into the Storage Stack

Raja Appuswamy
Vrije Universiteit, Netherlands
raja@cs.vu.nl

David C. van Moolenbroek
Vrije Universiteit, Netherlands
dcvmoole@cs.vu.nl

Andrew S. Tanenbaum
Vrije Universiteit, Netherlands
ast@cs.vu.nl

***Abstract*—Over the past few years, hybrid storage architectures that use high-performance SSDs in concert with high-density HDDs have received significant interest from both industry and academia, due to their capability to improve performance while reducing capital and operating costs. These hybrid architectures differ in their approach to integrating SSDs into the traditional HDD-based storage stack. Of several such possible integrations, two have seen widespread adoption: Caching and Dynamic Storage Tiering.**

Although the effectiveness of these architectures under certain workloads is well understood, a systematic side-by-side analysis of these approaches remains difficult due to the range of design alternatives and configuration parameters involved. Such a study is required now more than ever to be able to design effective hybrid storage solutions for deployment in increasingly virtualized modern storage installations that blend several workloads into a single stream.

In this paper, we first present our extensions to the Loris storage stack that transform it into a framework for designing hybrid storage systems. We then illustrate the flexibility of the framework by designing several Caching and DST-based hybrid systems. Following this, we present a systematic side-by-side analysis of these systems under a range of individual workload types and offer insights into the advantages and disadvantages of each architecture. Finally, we discuss the ramifications of our findings on the design of future hybrid storage systems in the light of recent changes in hardware landscape and application workloads.

I. Introduction

Over the last decade, flash-based SSDs (Solid State Disks) have revolutionized the storage landscape. Though modern flash SSDs perform much better than their rotating media counterparts under both random and sequential workloads, flash-only storage installations continue to be prohibitively expensive for most, if not all, enterprises due to the high cost/GB of SSDs. As a result, storage researchers have proposed implementing systems based on hybrid storage architectures that use high performance flash SSDs in concert with high-density HDDs (Hard Disk Drives) to reduce capital and operating costs, while improving overall performance.

Of all such architectures, two have gained widespread adoption—*Caching* [4] and *Dynamic Storage Tiering* (DST) [2], [12], [15]. The Caching architecture involves extending the two-level memory hierarchy to the third level by using flash devices as intermediate caches that sit between HDDs and memory. The DST architecture, on the other hand, uses SSDs for primary data storage by establishing tiers of high-performance flash storage and high-density disk storage.

Due to their popularity, these two architectures have also been in the limelight of research over the past few years, and the effectiveness of Caching and DST under certain specific workloads is well understood [9], [14]. However, with the wide spread adoption of storage virtualization, modern storage installations blend I/O requests from different workloads together into a single stream. Designing efficient hybrid architectures for such workloads requires answering two important questions: 1) how do existing architectures fare under such workloads?, and 2) should future hybrid systems support not one, but multiple architectures, and pair workloads with their ideal architectures?

In order to answer these questions, we need to perform 1) a side-by-side comparison of existing architectures under such mixed workloads, and 2) a systematic study of interactions between architectural design alternatives and workload parameters. Unfortunately, due to the wide range of configuration parameters and design alternatives involved in building Caching and DST-based systems, performing such a study would be infeasible in the absence of a hybrid storage framework.

In this paper, we will show how the Loris storage stack, with a few extensions, can be transformed into a modular framework for implementing and evaluating hybrid storage systems. To illustrate the flexibility of the framework, we will implement several flavors of Caching and DST. We will then use several macrobenchmarks and file system workload generators to perform a systematic study of the effectiveness of these Loris-based hybrid systems under a variety of workloads. Based on our evaluation, we will offer insights into 1) the design of current hybrid systems by investigating design factors that impact performance, and 2) the design of future systems in light of recent changes in hardware landscape and application workloads.

The rest of the paper is organized as follows. In Sec. II, we will present a classification of Caching and DST architectures based on several design parameters. In Sec. III, we will introduce the Loris stack and describe the plugin-based extensions that transform it into a hybrid storage framework. Following this, we will describe how we used this framework to implement Loris-based Caching and DST systems in Sec. IV. We will then present our side-by-side evaluation of these hybrid systems using several benchmarks in Sec. V. Finally, we will discuss the ramifications of our findings in Sec. VI, and conclude in Sec. VII.

II. Hybrid storage systems

As we mentioned earlier, Caching and DST architectures differ in the way they integrate SSDs into the HDD-based traditional storage stack. In this section, we will explore the design space of these hybrid architectures and classify them based on several design parameters.

A. Caching

Caching architectures use SSDs as a non-volatile, intermediate caches between the system memory (RAM) and HDDs. Thus, in all Caching architectures, SSDs contain only cached copies of HDD-resident primary data.

Based on when data is cached, Caching architectures can be classified as *On-demand* or *Interval-driven*. While data is cached as a side effect of a read operation is On-demand Caching, Interval-driven Caching monitors data blocks and periodically, once every preconfigured interval, trades old SSD-resident "cold" data for new HDD-resident "hot" data.

Irrespective of when data is cached, Caching architectures can be classified as *read-only* or *read-write* caches depending on their behavior with respect to write operations. Read-only caches maintain only clean data. Thus, writes to uncached data blocks are not buffered by the SSDs, and updates to cached data blocks invalidate the cached copies. ZFS's L2ARC [17] and NetApp's FlashCache [4] are examples of read-only caches used to speed up workloads dominated by random reads.

Read-write caches, on the other hand, cache both data reads and writes. They can be further classified into *Write-back* and *Write-through* caches. A Write-back cache eliminates all foreground HDD writes by buffering them in the SSD and resynchronizing the primary HDD copy later. Since the cached SSD copy and primary HDD copy can be out of sync in a Write-back Caching system, extra bookkeeping is required to maintain consistency and prevent data loss across power failures or system reboots. EMC's FastCache [5] is an example of a Write-back cache that uses flash drives configured as RAID1 mirror pairs to guarantee reliability in the face of system or power failures.

A Write-through cache, on the other hand, forwards writes to both the cached SSD copy and the primary disk copy. By maintaining all data copies in sync, Write-through caching avoids additional (potentially synchronous) metadata updates at the expense of foreground write performance. One could further classify a Write-through cache into a *Write-through-all* cache or *Write-through-update* cache depending on how writes to uncached blocks are handled. While a Write-through-all cache admits uncached data blocks, a Write-through-update cache sieves new data by admitting only cached data writes. Azor [14] is an example of a Write-through Caching system that supports both Write-through-all and Write-through-update Caching.

B. Dynamic Storage Tiering

Dynamic Storage Tiering architectures (DST) organize high-performance, flash-based SSDs and high-density, magnetic HDDs into multitier systems and partition data between tiers depending on several price, performance, or reliability factors. Thus, unlike Caching architectures, each data item in a DST system is stored in only one location.

Based on the initial allocation policy used, DST architectures can be classified into *Hot-DST* and *Cold-DST* types. With Hot-DST architectures, data is initially allocated on the HDD tier. Periodically, "hot" data are migrated to the SSD tier. With "Cold-DST" architectures, data are initially allocated on the SSD tier and "cold" data are periodically demoted to the HDD tier. IBM's EasyTier [19], Compellent's tiering systems [3], EDT [9] and HyStor [8] are a few examples of Hot-DST systems. Hot-DST architectures can be further classified depending on the time at which "hot" data are migrated. In *Dynamic Hot-DST* systems, "hot" data from the HDD tier are migrated on demand, while in *Interval-driven Hot-DST* systems, data are migrated at predefined intervals. Almost all Hot-DST systems we are aware of are interval driven.

At a very high level, Hot-DST and Caching architectures appear to be identical with respect to their mode of operation. Both Interval-driven architectures migrate/cache data at periodic intervals. Both On-demand architectures migrate/cache data as a side effect of a read operation. Furthermore, in order to be able to map data to their ideal storage targets, both Caching and DST architectures observe access patterns and classify data as "hot" or "cold." For instance, all Caching architectures use a second-level caching algorithm (like L2ARC [17]) and all DST architectures use some "hot" data identification mechanism (like inverse bitmaps [8]), to identify "hot" data that must be serviced by the SSDs. This raises two questions: 1) can popular DST algorithms be used for implementing efficient Caching architectures and vice versa?, and 2) all other factors considered identical, is there a performance impact associated with the most important design difference—presence or absence of a data copy?

Later, in Sec. IV, we will address the first question by showing how we use a popular DST algorithm to implement efficient Caching systems. Then, in Sec. V, we will evaluate the Caching and DST implementations side by side to answer the second question. Having described several hybrid architectures, we will now give a brief overview of the Loris stack and show how we use it as framework to implement hybrid systems.

III. Background: The Loris storage stack

In prior work, we proposed Loris [6], a redesign of the storage stack. Loris is made up of four layers as shown in Figure 1. 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 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.

A. Physical layer

The physical layer is tasked with providing 1) device-specific layout schemes, and persistent storage of files and their attributes, 2) end-to-end data verification using parental checksumming, and 3) fine-grained data sharing and individual file snapshotting. Thus, the physical layer exports a snapshotable physical file abstraction to the logical layer. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*.

B. Logical layer

The logical layer provides 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. 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 two sublayers. We will now briefly describe the functionalities of each sublayer.

The volume management sublayer is responsible for providing both file volume virtualization and per-file RAID services. It maintains data structures that track the membership of files in file volumes and mapping between physical files and logical files. It also provides file management operations that enable snapshoting and cloning of files or file volumes. In prior work, Loris has been used to design a new storage model [7]. File pools simplify storage administration and enable thin provisioning of file volumes [7]. 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.

Each file volume is represented by a *volume index file* that tracks logical files belonging to that volume. The volume index is created during volume creation, and it stores an array of entries containing the *configuration information* for each logical file in that volume. This configuration information is 1) the RAID level used, 2) the stripe size used, and 3) the set of physical files that make up the logical file. Similar to the way files are tracked by the volume index file, file volumes themselves are tracked using the *meta index file*. This file also contains an array of entries, one per file volume, containing *file volume metadata*. Thus, using these two data structures, the volume management sublayer supports file volume virtualization. Multiple file volumes can be created in a single file pool in Loris which makes thin provisioning of file volumes possible.

C. Cache and Naming layers

The cache layer provides data caching. As the cache layer is file-aware, it can provide different data staging and eviction policies for different files or types of files.

The naming layer acts as the interface layer. Our prototype naming layer implements the traditional POSIX interface. The naming layer uses Loris files to store data blocks of directories that contain directory entries. It also uses the attribute infrastructure in Loris to store POSIX attributes of each file as Loris attributes. All POSIX semantics are confined to the naming layer. For instance, as far as the logical layer is concerned, directories are just regular files.

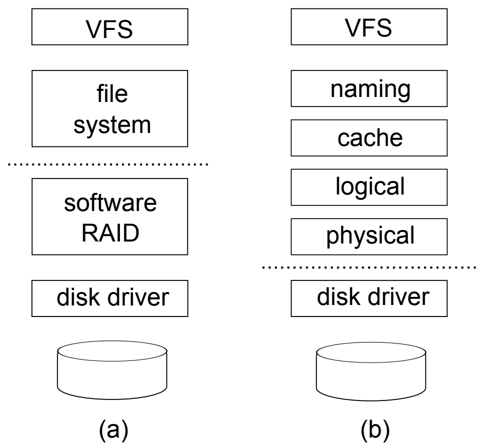

Figure 1: This figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

D. Tiering Framework

All hybrid storage systems, irrespective of how they integrate flash into the storage stack, essentially attempt to pair data with their ideal storage device to maximize performance. In order to do so, all these systems have to 1) collect and maintain access statistics to classify data, and 2) implement background migration to transparently relocate data to their designated target. We had to extend Loris to support these two functionalities in order to transform it into a framework.

We did this by extending Loris' logical layer. There were three main reasons for extending the logical layer compared to other layers. First, the logical layer uses the logical file abstraction to implement per-file RAID algorithms by multiplexing requests across physical files. We can exploit the same abstraction to support transparent migration of files between devices/physical modules. Second, the logical layer has information about both file access patterns and device performance characteristics, making it the ideal spot for implementing algorithms that take into account both these factors. Third, access statistics collected at the logical layer reflect the real storage workload after caching effects have been filtered out. Thus, by confining changes to the logical layer, we modularly extend the Loris stack to implement hybrid systems without affecting algorithms in any of the other layers.

1) Data collection plugin: Several DST and Caching systems have proposed collecting different access statistics for classifying data. For instance, EDT [9] is a DST system designed for installations that consist of SSD, SAS (Serial Attached SCSI) and HDD tiers. For achieving optimal performance under such multitier installation, EDT classifies hot extents into IOPS-heavy and bandwidth-heavy types, and stores IOPS-heavy extents on the SSD tier and bandwidth-heavy extents on the SAS tier. Thus for implementing an EDT-style DST system, one must collect and maintain IOPS and bandwidth requirements for each file. Azor [14], on the other hand, is an SSD-based Caching system that maintains access frequencies for each cached SSD block and uses it to perform cache admission control. Thus for implementing an Azor-style Caching system, one must maintain access counts for each file.

To support multiple such design alternatives, we extended the logical layer using a generic plugin model. The *data collection plugin* is responsible for collecting and maintaining access statistics for each file. Each plugin implementation is required to support a standard set of callback routines. During the startup phase, depending on the type of hybrid configuration to be deployed, the appropriate data collector is registered with the logical layer, which then invokes the callback routines at strategic points during execution.

We implemented a data collection plugin that uses inverse bitmaps [8] to identify performance-critical files that should be cached or migrated. During every read and write operation, the inverse bitmap b is calculated as shown below.

$$b = 2^{6-\lfloor \log_2(N) \rfloor} \tag{1}$$

In the equation, N refers to the number of 4-KB pages read/written from the file. The value 6 is an implementation-specific constant chosen based on the maximum number of 4-KB file pages read or written by the cache layer in a single operation (64). The computed value is then added to a 32-bit counter associated with that file. Thus, the inverse bitmap assigns a large weight to files read/written in small chunks, which could either be small files or large files randomly read/written in small chunks, thereby prioritizing random accesses over sequential ones.

Our current prototype maintains an in-memory array of counters, one per tier. When queried for the “hottest” file in the disk tier, the plugin picks the most recently used file with the highest counter value. When queried for the “coldest” file in the SSD tier, the plugin picks the least recently used file with the lowest counter value. Thus, we extend the original inverse bitmap design [8] by using recency as a tiebreaker among files with identical counter values. The in-memory approach is probably not scalable as modern installations consist of millions of files, so we are currently considering using priority dequeues using external heap variants or dynamic histograms with delayed updates for scalable maintenance of access statistics.

Our data collection plugin also explicitly keeps track of the counter value of the last file that has been evicted from the SSD tier during cleanup. It uses this value to perform admission control. A file is qualified for migration to the SSD tier only if its counter value is higher than that of the last evicted file.

We would like to point out here that all hybrid architectures we present in Sec. IV use the inverse bitmap plugin as their data collector. Thus, although inverse bitmaps were originally introduced and used in Hystor [8] for DST, we show how it can also used to implement high performance Caching architectures. Thus, as we mentioned earlier, most data collection algorithms are architecture neutral.

We would also like to point out that although we do not consider multitier installations (such as ones including SAS drives) as a part of this work, extending Loris to such configurations only requires replacing relevant plugins.

2) File migration: As we mentioned earlier, we exploited the logical file abstraction of the logical layer to support transparent file migration between physical modules. Our current implementation locks each file during migration to prevent foreground requests from accessing the source during migration. We are also working on implementing transparent, incremental migration of file data. The incremental migration implementation would first take a snapshot of the target file using the individual file snapshoting functionality present in Loris, following which it would copy the snapshot’s data and attributes to the designated target. After successfully copying the snapshot, the migration plugin would then, if need be, perform an incremental transfer of data modified since the snapshot.

E. Loris as a platform for storage tiering - The Pros

There are several advantages in using Loris as the basis for implementing DST solutions. First, most DST solutions exploit device heterogeneity to improve performance. For instance, Avere’s DST system [2] stores all write-only files on the SAS tier using a log-structured layout to optimize write throughput. Since the Loris stack provides the capability to pair devices with their ideal layout algorithms, it can be used to exploit heterogeneity inherent in tiered systems.

Second, several DST systems use semantic information to identify crucial data (like file system metadata). As we mentioned earlier, semantic information is exchanged between layers in the Loris stack using the attribute infrastructure. In the Loris stack, each file create carries with it a file type attribute that informs the logical layer if the file is a metadata file (directory) or a data file. Thus, the logical layer can use this semantic information to assign different policies to files or file types. We will show later how we use semantic information to implement 1) *type-aware sieving* of large files, and 2) per-file tiering policy later in the paper.

Third, administrative operations like hot-swapping and online addition and removal of devices are mandatory features in any enterprise DST system. The file pool model in Loris simplifies storage administration and is capable of supporting all these features.

F. Loris as a platform for storage tiering - The Cons

Since Loris’ logical layer maintains mapping information at the granularity of whole files, implementing Caching or DST systems that operate on a sub-file basis is not possible. Consider an append write to an uncached file for instance. In order to implement Write-back Caching, Loris would have to buffer this write in the SSD. Doing so would require the logical layer to map two sets of logical file offsets to two different physical files (offset range <0, old file size -1> to physical file stored in HDD, and range <old file size, new file size -1> to a physical file on the SSD). The current mapping infrastructure only supports mapping a whole logical file to one or more physical files.

However, we would like to emphasize the fact that this is a limitation of just the current implementation. We are working on designing a new mapping format for the logical layer that supports sub-file mapping. With the new infrastructure, Loris would select the mapping type on a per-file basis. For instance, while all small files and files read/written in their entirety could be stored in a mapping file based on the old format, a file that is read/written in 4-KB chunk could use the new format that could potentially map each 4-KB logical block to a different physical file. By using this extension mapping infrastructure, we intend to evaluate block, extent and file-level tiering and Caching implementations side by side as a part of future work.

We would also like to point out that despite the lack of subfile mapping capability in Loris, all the results we present in this work are equally applicable to block or extent-level implementations. As all Caching and DST architectures are implemented using a single framework, and as all of them share the same data collection plugin (which maintains access statistics at the granularity of whole files), we believe that a block or extent-based realization of these architectures, under similar workloads, using the same access statistics would produce comparative results identical to our study.

IV. Loris-based hybrid systems

Having described the plugins, we will now show how we use these plugins to implement several Caching and DST systems for a two-tier (SSD/disk) installation. Common to all these systems is the *type-aware sieving* of large files. During preliminary evaluation of these hybrid systems, we found out that certain benchmarks (Web Server) create large, append-only log files that were never read. As these files received a lot of writes, their bitmap counter values were high. As a result all hybrid systems pinned these log files to the SSD tier, thereby wasting valuable space that could be used for housing other "genuinely hot" files. To prevent this, we added type-aware sieving to Loris. With sieving, any file larger than a configurable threshold, which in our current prototype is 1-MB, will not be cached or migrated to the SSD. Similar, any SSD-resident file is explicitly demoted (in the case of DST) or invalidated (in the case of Caching) when it grows beyond 1-MB. Type-aware sieving is an example of how we use semantic awareness of the Loris stack to improve the performance of all hybrid systems.

Also common to all these systems is the cleaner implementation. A cleanup of the SSD tier is triggered when a write operation to the SSD tier cannot be completed due to lack of space. This can happen either during a foreground write operation to a file in the SSD tier, or during background migration/caching of a file from the disk tier. In both cases, the cleaner consults the data collection plugin to determine the set of cold files to evict from the SSD tier. The action taken by the cleaner depends on the architecture being implemented. For Caching architectures, the cleaner simply invalidates the cached file copy by deleting it from the SSD. For DST implementations, the cleaner invokes the migration plugin, demoting those files back to the disk tier. This cold migration continues until enough space has been cleared to finish the write operation. In addition to such foreground cleaning, we also run the cleaner in the context of a background thread, to proactively clean the SSD tier, under certain hybrid configurations as we will show later.

We will now describe how we implemented several hybrid storage systems using the Loris stack.

A. Loris-based Hot-DST systems

We will now describe the Loris-based implementation of two Hot-DST architectures. As explained in Sec. II, all Hot-DST architectures allocate data on the HDD tier. They differ based on when they migrate "hot" data to the SSD tier.

1) Dynamic Hot-DST: Our Dynamic Hot-DST implementation migrates "hot" files as a side effect of a read operation that is serviced by the HDD tier. It first queries the data collection plugin to determine if the file is a valid migration candidate. As we mentioned earlier, our data collection plugin considers a file to be a valid candidate if its counter value is higher than that of the file last evicted from the SSD tier. In such a case, the DST implementation queues the file for migration with the migration plugin.

2) Interval-driven Hot-DST: We also implemented a system based on the Interval-driven Hot-DST architecture. Every preconfigured number of seconds, our Hot-DST implementation runs in the context of a background thread and migrates "hot" files identified by the data collection plugin to the SSD tier. Hot file migration continues until all potential candidates have been migrated or the "hottest" file in the disk is colder (has a lower counter value) than the "coldest" file in the SSD tier. The data collection plugin verifies the latter condition by comparing the next candidate file's access counter with that of the file last evicted from the SSD.

As the interval of migration is a configuration parameter, we will use two versions (five and eighty seconds) of our Interval-driven Hot-DST system to evaluate the impact of migration interval on overall performance.

B. Loris-based Cold-DST architectures

We will now describe the Loris-based implementation of three Cold-DST architectures. As explained in Sec. II, all Cold-DST architectures allocate data on the SSD tier.

1) Plain Cold-DST: This is conceptually the simplest of all DST implementations. This system does not perform any form of "hot" file migration. Foreground write requests to files in the SSD tier that are unable to complete due to lack of space automatically trigger tier cleanup. Files that are demoted during cleaning are never migrated back to the SSD tier, not even if they become "hotter" at a later point in time. Thus, this implementation uses the data collection plugin only to determine which files to evict from the SSD tier.

2) Dynamic Cold-DST: While the Plain Cold-DST system would work well with workloads where newly created data accounts for a significant fraction of accesses, it would perform poorly under workloads with shifting locality. This is because any access to data that has been "cold" migrated will be serviced by the disk tier.

We solve this problem by adding on-demand "hot"-file migration to the Plain Cold-DST system. Similar to the Dynamic Hot-DST system, data-collector-approved files are migrated in the background as a side effect of a read operation that finds the file in the HDD tier. However, unlike the Hot-DST counterpart, new files continue to be allocated on the SSD tier in the Dynamic Cold-DST system.

3) Dynamic Cold-DST with background cleaning: When operating with a full SSD, dynamic migration and foreground write requests trigger tier cleanup. As cleaning requires migrating "cold" files off the SSD tier, these writes blocks until sufficient free space has been generated. To avoid stalling write requests, we added a proactive background cleaner to our Dynamic Cold-DST implementation. The cleaner implementation maintains a running counter of the total amount of "cold" data evicted from a full SSD tier as a side effect of foreground or dynamic migration writes. It uses this counter as an estimate of the amount of space to recover for speeding up future write operations. The cleaner runs in the context of a background thread and starts evicting "cold" files as a side effect of the first blocking write request.

C. Loris-based Caching

As we explained earlier, the whole-file mapping infrastructure of our current prototype makes it impossible to implement Write-back or Write-through-all Caching systems. We will now describe the implementation of two Write-through-update Caching architectures.

1) On-demand Caching: On-demand Caching, for most part, works similar to Dynamic Hot-DST system. with the only difference being the fact that files are cached rather than being migrated. As the primary file copy continues to reside in the HDD, unlike the Hot-DST counterpart, cleaning the SSD only requires deleting "cold" files to invalidate them. As cleaning can happen as a side-effect of foreground write operation, and as On-demand Caching is identical to Dynamic Hot-DST in every other aspect, we can compare the two implementations head-to-head to measure the SSD cleaning overhead.

2) Interval-driven Caching: Interval-driven Caching works identical to the Interval-driven Hot-DST implementation with the exception that files are copied rather than migrated. Similar to its Hot-DST counterpart, all files are initially allocated on the HDD. Periodically, at a configurable interval (five seconds in our current prototype), the statistics accumulated by the data collector are used to cache "hot" files in the SSD tier. Similar to the Dynamic Hot-DST—Caching comparison, we can also compare the two Interval-driven architectures to measure the impact of cleaning on overall performance.

V. Evaluation

Having described how we implemented various hybrid architectures, we will now present a systematic analysis of the effectiveness of these architectures under a wide range of workloads. We will first describe the hardware setup and benchmarking tools we used for our evaluations. We will then present a side-by-side evaluation of these architectures and offer insights into the interaction between design alternatives and workload parameters.

A. Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4-GB RAM, using a 500-GB 7200-RPM Western Digital Caviar Blue SATA hard disk (WD5000AAKS), and a OCZ Vertex3 Max IOPS SSD. Table I lists the performance characteristics of these devices. We ran all tests on 8-GB test partitions at the beginning of the devices.

Device	*BB/s*	*Random 4K IOPS*
WD5000AAKS	126/126	112/91
OCZ Vertex 3 MAX IOPS	550/500	35000/75000

Table I: Device properties: The table lists the read/write performance characteristics of the two SATA devices we use for evaluating various hybrid architectures.

The Loris prototype has been implemented on the MINIX 3 multiserver operating system [13]. We deliberately configured Loris to run with a 64-MB data cache to ensure that the working set generated by benchmarks is at least an order of magnitude larger than the in-memory cache. Thus, using a low cache size, we heavily stress the I/O subsystem.

To estimate the effect that SSD size has on the effectiveness of various DST models, we modified the Loris stack to keep track of the amount of user data written to the SSD and used it to artificially limit the available SSD size. We used this to evaluate the effectiveness of various architectures at three different SSD sizes, namely, 25%, 50%, and 75% of the total working set size. We determined the working set size by running each workload using a disk-only configuration.

We are aware of the fact that certain consumer grade SSDs exhibit performance deterioration when occupied at 100% capacity. Although our artificial approximation of available SSD size sidesteps this issue, we believe that the results we derive are still applicable as we target enterprise installations that are likely to use enterprise-grade SSDs. Unlike consumer-grade SSDs, these high-end SSDs are known to overprovision large amounts of scratch space to avoid the write-cliff phenomenon [11].

B. Benchmarks and Workload Generators

Since we wanted to systematically analyze the interactions between architectural design alternatives and workload parameters, we used Postmark and FileBench to generate four different classes of server workloads. Postmark is a widely used, configurable file system benchmark that simulates a Mail Server workload. It performs a specified number of transactions, where each transaction pairs a whole-file read or append operation with a create or delete operation. We report the transaction time, which excludes the initial file preallocation phase, for all hybrid systems.

FileBench is an application-level workload simulator that can be used to model workloads using a flexible workload modeling language (WML). We used two predefined workload models to generate File Server and Web Server workloads. We ran each workload for half an hour and we present the IOPS reported by FileBench for all hybrid systems. Preliminary evaluation revealed wide variations in results across different FileBench runs (even with the same hybrid architecture). We traced this back to the variation in random seed selection between runs. In order to reliably compare different hybrid architectures, we modified FileBench to use a fixed random seed across all runs. With this patch, all the results we obtained were reproducible.

C. Workload Categories

Using Postmark and FileBench, we were able to vary a variety of workload parameters like file size distribution, read-write ratios, access patterns (sequential vs. random) and access locality (random vs Zipf). We will now detail the properties and configuration parameters of each workload.

1) Mail Server: We configured Postmark to perform 80,000 transactions on 40,000 files, spread over 10 subdirectories, with file sizes ranging from 4-KB to 28-KB, and read/write granularities of 4-KB. The resulting workload is dominated by small file accesses, has a 1:2 read-write ratio, and exhibits random access pattern with very little locality.

2) File Server: We configured FileBench to generate 10,000 files, using a mean directory width of 20 files. The median file size used is 128-KB, which results in files an order of magnitude larger than the rest of the workloads. 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. As all files are read in their entirety, and as append operations are large, the access pattern is sequential. The workload lacks locality as file access distribution is uniform.

3) Web Server: 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 only 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. This results in a 10:1 read-write ratio unlike other benchmarks. Though files are read in their entirety, the small file size results in the file access pattern being essentially random. Similar to the File Server workload, this workload also lacks locality due to the uniform file access distribution.

4) Web Server (Zipf Distribution): All the workloads described above lack locality which is present in several real-life workloads. For instance, while the Web Server workload generated by FileBench lacks locality, it is well known that file accesses in web servers tend to follow a Zipf distribution [10]. Thus, we modified FileBench to generate a Zipf-based file access distribution. In addition, we also wanted to evaluate the effectiveness of Dynamic Cold-DST architecture under workloads with shifting locality. To do so, we modified the default Web Server workload to generate two sets of 25,000 files instead of one. As a result, the first fileset get flushed out to the HDD tier as a part of cleaning up the SSD tier to accommodate the second one. We then ran the workload generator on the first set, thereby simulating shifting workload locality - the adversarial case for the Plain Cold-DST architecture.

D. Comparative evaluation

Having described the workload parameters, we will now present our evaluation of various hybrid systems. We will first analyze Loris-based DST systems to identify the impact of architecture-specific design alternatives (Sec. V-D1, Sec. V-D2). We will then compare DST systems with their Caching counterparts to identify top performers under various workloads (Sec. V-D3). Finally, we will present the side-by-side comparison of these top performers under a mixed workload that simulates a virtualized workload (Sec. V-E).

1) Hot-DST Architectures: Figures 2, 3, 4, 5 show the performance of various Hot-DST configurations. There are a number of interesting observations to be made from the results.

First, as can be seen in Figure 5, locality plays a major role in deciding the effectiveness of Hot-DST architectures. Even at SSD sizes covering as little as 25% of the total working set, all Hot-DST configurations show significant performance improvement compared to the HDD-only case.

Second, under all workloads, the migration interval has a significant impact on overall performance. For instance, under the Web Server workload (Figure 4), the five-second, Interval-driven Hot-DST system delivers lower IOPS than even the HDD-only case. The eighty-second Hot-DST system, on the other hand, improves performance significantly. As we mentioned earlier, the Web Server and File Server workloads access files uniformly in sequence without any locality. As a result, when we traced migration patterns at the logical layer, we found quite a large number of files shuttling back and forth between tiers, with the number of such files increasing as the migration interval decreases. Thus, the thrashing of files caused by the workload's uniform access pattern destroys performance by interfering with foreground reads and writes serviced by the HDD tier.

Third, unlike other workloads, the eighty-second, Interval-driven Hot-DST performs poorly under Postmark. At low SSD sizes, it takes longer than the HDD-only configuration to finish the transactions. Even at high SSD sizes covering as much 75% of the working set, it performs only slightly (12%) faster. Analysis revealed that this was due to two factors. First, as we mentioned earlier, the workload generated by Postmark lacks locality. Thus, it is adversarial in nature for locality-dependent algorithms like Interval-driven Hot-DST. Second, unlike FileBench, Postmark is transaction-bound rather than time-bound (the termination condition is a limit on the number of transactions). Although we perform 80,000 transactions, analysis revealed that the total transaction time was not long enough for Interval-driven Hot-DST to stabilize.

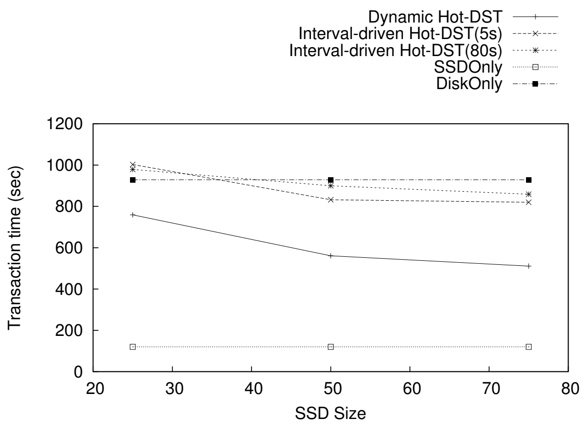

Figure 2: Transaction time (seconds) under Postmark for various Hot-DST architectures

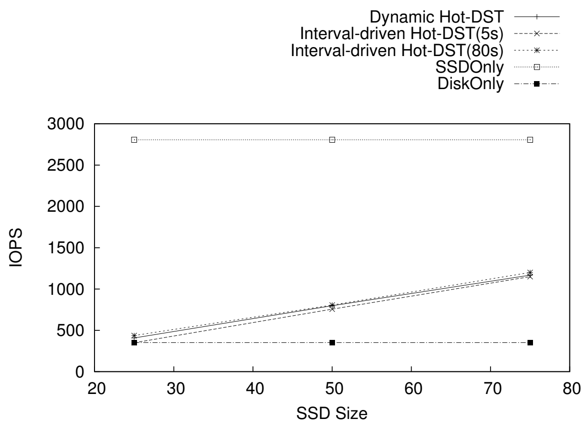

Figure 3: IOPS delivered under FileBench's File Server workload by various Hot-DST architectures

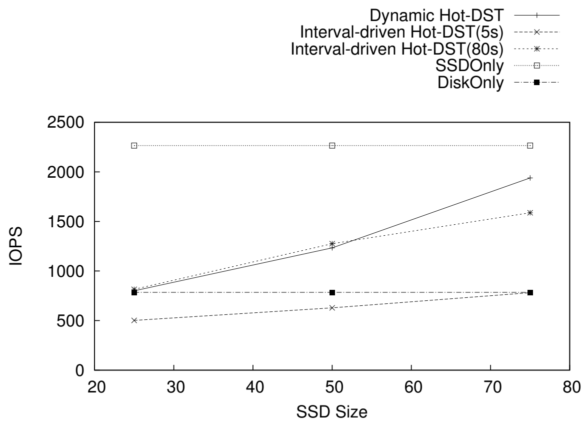

Figure 4: IOPS delivered under FileBench's Web Server workload by various Hot-DST architectures

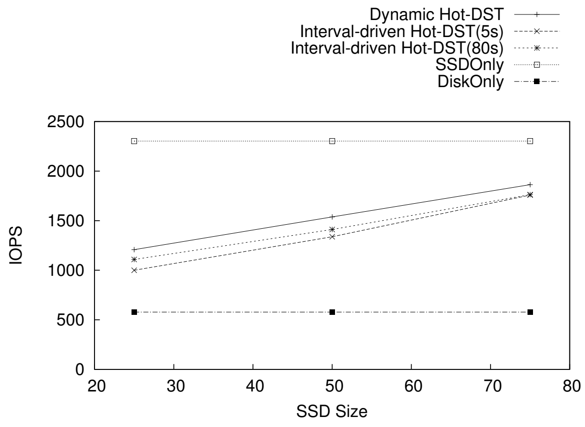

Figure 5: IOPS delivered under FileBench's Web Server (with Zipf) workload by various Hot-DST architectures

Fourth, while Dynamic Hot-DST performs similar to other Hot-DST architectures under File Server (Fig. 3) and Web Server (Fig. 4) workloads, it provides significant improvement under Postmark (Fig. 2). This is unexpected, especially given the fact that the aforementioned reasons that slow down Interval-driven Hot-DST also apply to Dynamic Hot-DST. By profiling the system, we found the source of this performance improvement to be a counterintuitive increase in the number of writes serviced by the SSD tier under Dynamic Hot-DST. As we mentioned earlier, Postmark pairs creates/deletes with reads/writes. Writes issued by Postmark append data to existing files. As these append operations are typically not block aligned, they trigger an "append-read" operation to fetch the append target (the file's last data block). The ensuing "append write" is then buffered by the data cache (caching layer). As a side effect of this read operation, the target file's access counter gets updated by a large increment (due to the small amount of data being read), which results in the file being migrated to the SSD. When the data is flushed from the cache at a later time, it gets serviced by the SSD (which now hosts the "hot" file) resulting in the performance improvement. Interval-driven DST architectures do not benefit from these "append-reads" as they miss the window of opportunity due to not performing on-demand migration. We verified that this was indeed the case by not updating access statistics on append reads. Under such circumstances, the Interval-driven Hot-DST outperformed Dynamic Hot-DST.

2) Cold-DST Architectures: Figures 6, 7, 8, 9 show the performance of various Cold-DST configurations. As can be seen in Figure 9, the adversarial workload with locality results in Plain Cold-DST performing poorly when compared to the Dynamic Cold-DST architecture. An interesting observation is that the Plain Cold-DST architecture still performs noticeably better than the HDD-only case. On investigating this, we found that during the preallocation phase, directories receive quite a lot of read/write accesses. These accessess cause directories to possess high counter values in comparison to other files. As a result, despite the barrage of creates and writes during the preallocation phase, directories remain pinned to the SSD tier. Thus, all directory accesses during the workload run are serviced by the SSD causing a noticeable performance improvement over the HDD-only case.

If we consider workloads without locality, we see that the Dynamic Cold-DST architecture deteriorates performance compared to Plain Cold-DST. Due to the lack of locality, the performance gained by servicing reads from the SSD tier does not match the overhead of migrating "hot" data from the HDD tier. However, an interesting observation is how, despite similar access patterns (uniform without locality), File Server and Web

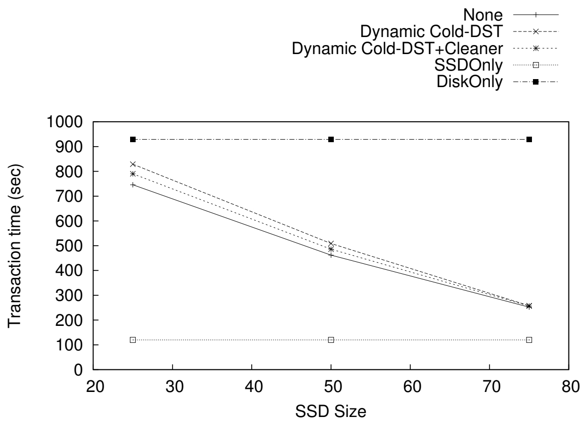

Figure 6: Transaction time (seconds) under Postmark for various Cold-DST architectures

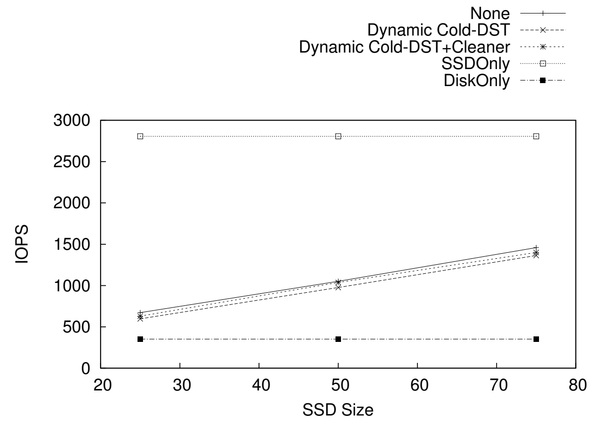

Figure 7: IOPS delivered under FileBench's File Server workload by various Cold-DST architectures

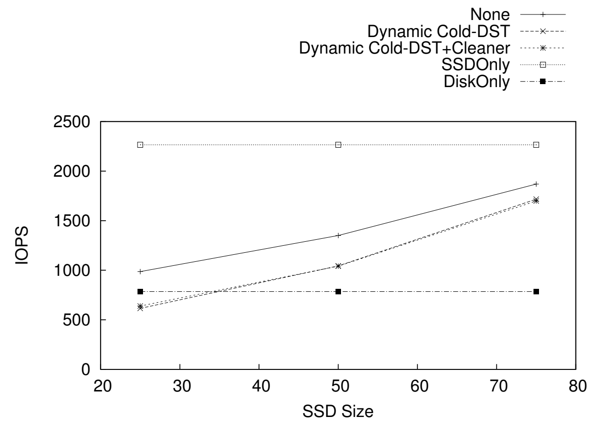

Figure 8: IOPS delivered under FileBench's Web Server workload by various Cold-DST architectures

Server workloads produce different comparative results. While Dynamic Cold-DST catches up with Plain Cold-DST under the File Server workload (Figure 7), it continues to lag behind by a huge margin under the Web Server workload (Figure 8).

On investigating this further, we found that the delete operations performed by the File Server workload play an indirect, albeit crucial, role in improving the overall performance of Cold-DST architectures. Logically speaking, creating new files in an already full SSD tier should have an adverse impact on performance as these writes cannot be completed without evicting "cold" data. However, in reality, newly written data is first buffered by Loris' data cache (cache layer) before being flushed out to the SSD. Unlike write operations, file deletes propagate down through the layers immediately. These delete operations free up space in the SSD tier indirectly accelerating both delayed foreground writes and background dynamic migrations. As the performance gained by allocating new files on the SSD tier offsets the performance drop caused by migrating "hot" files, Dynamic Cold-DST catches up with Plain Cold-DST under the File Server workload.

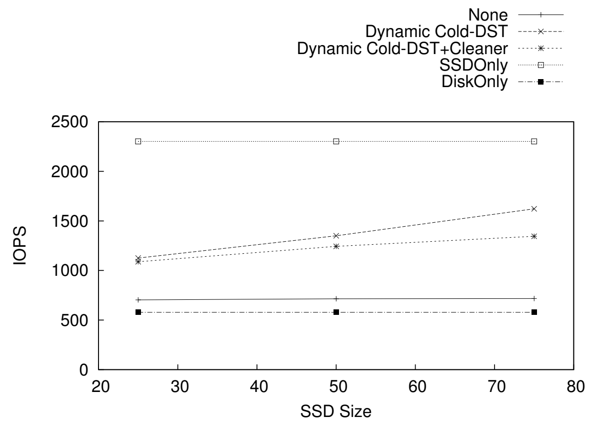

Figure 9: IOPS delivered under FileBench's Web Server (with Zipf) workload by various Cold-DST architectures

We saw only marginal improvement (at best) from adding background cleaning to the Dynamic Cold-DST architecture. We believe that this is due to our cleaner being overly conservative in freeing up space. During experimentation, we found out that aggressive cleaning had a negative impact on performance under most workloads we used in this study. However, recent analysis of network file system traces indicate that over 90% of newly created files are opened less than five times [16]. Under such conditions, a Dynamic Cold-DST implementation would definitely benefit from aggressive cleaning. We intend to perform a reevaluation of our Cold-DST implementations under trace-driven workloads as a part of future work.

3) DST vs Caching: Figures 10, 11, 12, 13 show the performance of the two Caching architectures. In addition, we also include the top performers from other architecture types (Dynamic Hot-DST, Interval-Driven Hot-DST (eighty-second), and Plain or Dynamic Cold-DST depending on the workload) so that we can perform a side-by-side comparison of Caching and DST architectures. Several interesting observations can be made from these figures.

First, as the Web Server workload with Zipf locality reveals (Figure 13), all hybrid configurations perform significantly better than the HDD-only case at all SSD sizes, thus, proving the effectiveness of hybrid architectures.

Second, On-demand Caching consistently outperforms Interval-driven Caching under all workloads. The same reasoning behind Dynamic Hot-DST being faster than its Interval-driven counterpart applies here as well—Interval-

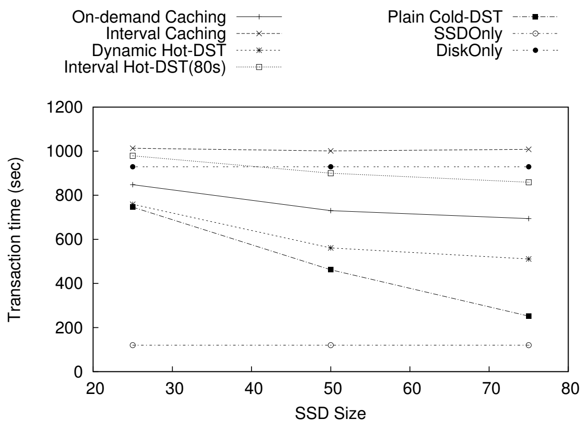

Figure 10: Transaction time (seconds) under Postmark for Caching and DST architectures

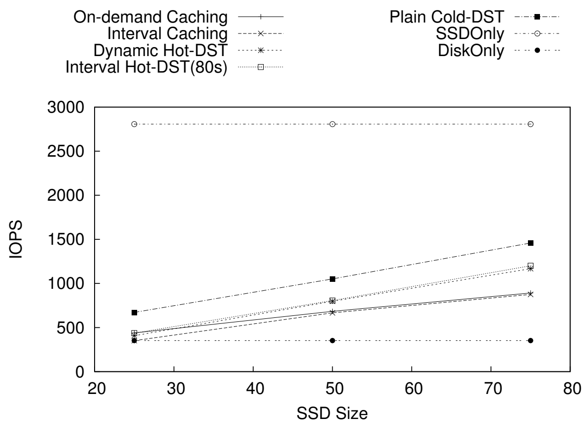

Figure 11: IOPS delivered under FileBench's File Server workload by Caching and DST architectures

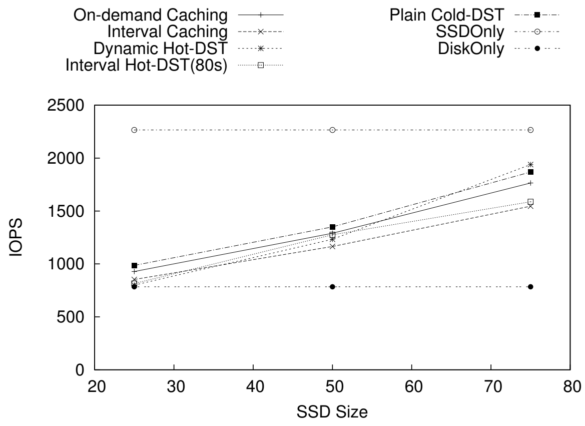

Figure 12: IOPS delivered under FileBench's Web Server workload by Caching and DST architectures

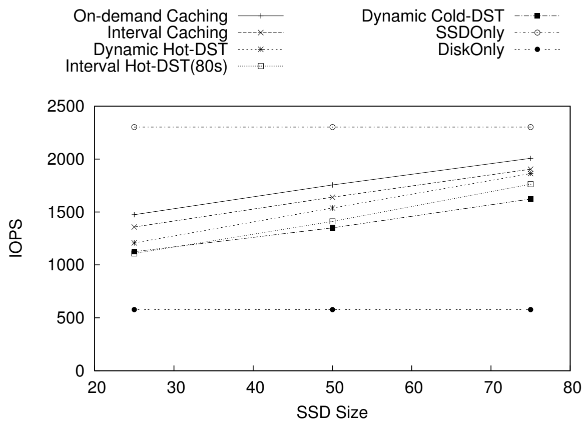

Figure 13: IOPS delivered under FileBench's Web Server (with Zipf) workload by Caching and DST architectures

driven Caching suffers from the same thrashing issues as its DST counterpart.

Third, Postmark and File Server workloads (Figures 10, 11) reveal the impact of an important design parameter—the presence or absence of a data copy. As we mentioned earlier, the DST and Caching implementations are identical in all aspects except for the fact that Caching implementation copies files while the DST implementation migrates them. Since we implemented Write-through-update Caching, any writes to these cached copies are also written through to their primary disk replica. Since Postmark and File Server workloads consist of append operations, all Caching architectures suffer due to the necessity to keep the two copies in sync. Thus, they perform consistently worse than their DST counterparts. This shows how Hot-DST architectures are preferable over their Caching counterparts under workloads with a low read:write ratio.

Fourth, the Web Server workload (Figures 12, 13) reveals a drawback inherent to DST. Unlike Postmark and File Server, we see that the Caching architectures outperform their DST counterparts at low SSD sizes. As we mentioned earlier, under the Web Server workload, the only write operations issued are those that append data to the log file. Since our implementation pins the log file to the HDD tier, SSDs are used for serving only reads. As a result, Caching architectures incur no consistency-related overhead. Furthermore, SSD tier cleaning under Caching architectures involves invalidating the cached copy by just deleting it. DST architectures, on the other hand, have to migrate "cold" data back to the HDD tier incurring an additional overhead. This shows how Caching architectures are preferable over their Hot-DST counterparts under workloads with a high read:write ratio.

Fifth, Cold-DST architectures meet or exceed the performance achieved by other architectures under all workloads except the Web Server workload with locality (Figure 13). We believe that this oddity is in large part due to the preallocation phase. As we explained earlier, FileBench first preallocates files before starting the workload generators. As Cold-DST architectures allocate files on the SSD tier, they start operating with a full SSD tier. Hot-DST and Caching architectures, on the other hand, start with a near-empty SSD tier as they perform initial allocation on the HDD tier. Thus, "hot" file migration under Hot-DST and Caching architectures incurs no cleaning overhead until the SSD tier gets full. However, dynamic migration under Cold-DST architectures incurs cleaning overhead from the very beginning causing a noticeable performance drop.

E. Mixed Workloads and Hybrid Architectures

In order to analyze the effect of storage virtualization on the performance of various hybrid architectures, we wrote a FileBench workload model that blends File Server and Web Server workloads into a single stream. We used the same configuration parameters as the individual workloads. In addition, we also preallocated a dummy fileset to simulate shifting locality by flushing valid data off the SSD.

1) Type-aware DST: In order to understand if pairing workloads with ideal architectures is better than adopting a "one-architecture-for-all" approach, we modified the Loris stack to support *Type-aware DST*. Our Type-aware DST implementation associates a tiering policy with each file volume (a rooted hierarchy of files and directories). We created two file volumes, one per workload, and tag volumes with policies that directed the logical layer to pair On-demand Caching with the Web Server workload and Dynamic Hot-DST with the File Server workload. Thus, our Type-aware DST implementation caches or migrates "hot" files depending on whether they belong to the Web Server or File Server volume.

Figure 14 shows the performance of individual Caching and DST architectures side-by-side with our Type-aware DST architecture under the mixed workload. There are three important observations to be made. First, Dynamic Cold-DST meets the performance of type-aware tiering at low SSD sizes, and exceeds it at higher sizes. Contrasting this with the performance of Dynamic Cold-DST under just the Web Server workload (Figure 8), we clearly see that the performance improvement achieved by allocating new files in the SSD tier overshadows the adverse effect of dynamic migration.

Second, the Caching configuration suffers under the mixed workload. This can be attributed to the synchronization overhead caused by append operations in the File Server workload.

Third, the Type-aware DST outperforms both individual architectures at all SSD sizes as it possesses the advantages of both Caching and Dynamic Hot-DST architectures without any of their disadvantages. By migrating files created by the File Server workload, and caching files created by the Web Server workload, Type-aware DST reduces the cleaning overhead without incurring the expensive synchronization overhead. This illustrates the benefit of pairing workloads with their ideal architectures. The Type-aware DST architecture we implemented is only one of many possible alternatives. For instance, one could also pair Cold-DST architecture with the File Server workload and Dynamic Hot-DST architecture with the Web Server workload. We intend to implement such architectures and evaluate them using file system traces as a part of future work.

VI. Discussion

Based on our experience designing and evaluating various hybrid architectures, we will now present a few open research problems that need to be solved in order to be able to design efficient hybrid storage architectures.

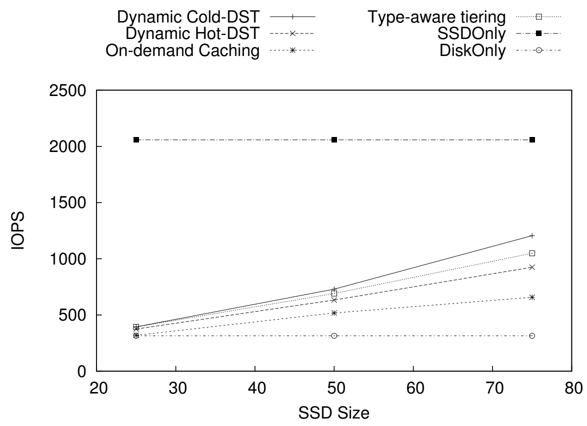

Figure 14: IOPS delivered under FileBench's mixed workload by Caching and DST architectures

A. Analyzing Cold-DST

While Hot-DST systems have received a lot of attention over the past years, we believe Cold-DST architectures have been ignored due to two main reasons, namely, poor write performance, and inferior reliability. The design of early hybrid systems that used first generation SSDs focused on improving the overall system performance by pairing read-only data with SSD tier and write-only data with the HDD tier. Researchers have pointed out that the limited lifetime (erasure cycles) of NAND-flash-based SSDs must be considered as a critical factor in the design of hybrid storage systems, and have even proposed using HDD-based logging to improve the longevity of SSDs [18]. As Cold-DST architectures write significantly more data to the SSD tier than other architectures, they will most certainly wear out these SSDs faster.

However, unlike first generation SSDs, which suffered from poor random write performance due to inefficient Flash Translation Layer (FTL) designs (among several other reasons), modern SSDs have exceptionally high write performance, sometimes even exceeding read performance [1]. Similarly, modern enterprise-grade, SLC flash-based SSDs have reasonably high reliability. For instance, OCZ Vertex2 EX SLC SSD [1] has an MTBF rating of 10 million hours.

In light of these recent changes in the storage hardware landscape, modern DST systems (like Hystor [8]) have started allocating dedicated write-back areas in SSDs to improve write performance. Cold-DST architectures are capable of meeting the performance offered by Write-back Caching without any of the synchronization-related performance issues. However, there are several important design factors that require further research. Should a Cold-DST implementation partition the SSD space into read and write areas, and if so, can it dynamically determine partition sizes? Recent analysis of network file system traces indicate that over 90% of newly created files are opened less than five times [16]. Can we utilize SSD parallelism to perform aggressive cold migration without affecting foreground accesses under such workloads?

B. Other hybrid architectures

In addition to DST and Caching architectures, there are several other ways SSDs could be integrated into the storage stack. For instance, SSDs could be used as dedicated data stores for housing specific data types. One such example is using SSDs for exclusively storing file system metadata, executables and shared libraries, as suggested by the Conquest file system [21]. Researchers have shown how MEMS-based storage can be used in several capacities to accelerate performance of disk arrays [20]. Similarly, SSDs could also be used in heterogeneous disk arrays to eliminate redundancy-related performance bottlenecks. To our knowledge, such configurations have not been compared side-by-side with DST or Caching architectures, and such a systematic study would help determine the best possible way to integrate SSDs into the storage stack.

C. Caching vs Tiering Algorithms

Earlier in this paper, we showed how inverse bitmaps, a "hot" data identification mechanism originally used to implement a DST system, can also be used to implement effective Caching architectures. The cross-architecture applicability of several data collection algorithms raises several research questions like 1) how effective are second-level buffer cache management algorithms when used to implement DST architectures?, 2) does the relative performance of various architectures remain unaffected across different data collection algorithms?

We intend to use the Loris stack to implement several hybrid architectures and answer these research questions as a part of future work.

VII. Conclusion

We showed how our plugin-based extensions to the Loris stack transform it into a framework for implementing hybrid storage solutions. Using the Loris framework, we illustrated the effectiveness of DST and Caching by showing how these hybrid architectures can outperform a disk-only configuration even with SSD sizes covering as little as 25% of the working set. Based on our evaluation, we offered several insights into interactions between architecture-specific design alternatives and workload parameters. We also discussed the ramifications of our work by highlighting a few areas that deserve more attention from storage researchers.

Acknowledgment

This work has been supported by the European Research Council Advanced Grant 227874.

References

[1] "OCZ Vertex 2 EX," http://www.ocztechnology.com/res/manuals/-OCZ_Vertex2EX_Product_sheet_1.pdf.

[2] "Avere Systems, The Avere Architecture for Tiered NAS," 2009.

[3] "Compellent Harnessing SSD's Potential," ESG Storage Systems Brief, 2009.

[4] "NetApp's Solid State Hierarchy," ESG White Paper, 2009.

[5] "EMC Fast Cache - A Detailed Review," EMC White Paper, 2011.

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

[7] 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 Systems and Tech.*, 2011.

[8] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems," in *ICS*, 2011.

[9] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami, "Cost Effective Storage using Extent Based Dynamic Tiering," in *Proc. of the Ninth USENIX Conf. on File and Storage Tech.*, 2011.

[10] A. Gulati, M. Naik, and R. Tewari, "Nache: Design and Implementation of a Caching Proxy for NFSv4," in *Proc. of the Fifth USENIX Conf. on File and Storage Tech.*, 2007.

[11] L. G. Harbaugh, "Storage Smackdown: Hard drives vs SSDs," http://www.networkworld.com/reviews/2010/041910-ssd-hard-drives-test.html, 2010.

[12] J. Harler and F. Oh, "Dynamic Storage Tiering: The Integration of Block, File and Content," 2010.

[13] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, "Construction of a Highly Dependable Operating System," in *Proc. of the Sixth European Dependable Computing Conf.*, 2006.

[14] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas, "Azor: Using Two-Level Block Selection to Improve SSD-Based I/O Caches," in *Proc. of the Sixth Intl. Conf. on Networking, Arch., and Storage*, 2011.

[15] B. Laliberte, "Automate and Optimize a Tiered Storage Environment - FAST! ESG White Paper," Tech. Rep., 2009.

[16] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller, "Measurement and Analysis of Large-Scale Network File System Workloads," in *Proc. of the USENIX Ann. Tech. Conf.*, 2008.

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

[18] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," in *Proc. of the eighth USENIX Conf. on File and Storage Tech.*, 2010.

[19] Taneja Group Technology Analysts, "The State of the Core Ű Engineering the Enterprise Storage Infrastructure with the IBM DS8000," White Paper, 2009.

[20] M. Uysal, A. Merchant, and G. A. Alvarez, "Using MEMS-Based Storage in Disk Arrays," in *Proc. of the Second USENIX Conf. on File and Storage Tech.*, 2003.

[21] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System," in *Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, 2002.