Write Policies for Host-side Flash Caches

Ricardo Koller,[§‡] Leonardo Marmol,[§] Raju Rangaswami,[§] Swaminathan Sundararaman,[†]
Nisha Talagala,[†] Ming Zhao[§]
[§]*Florida International University* [†]*Fusion-io* [‡]*VMware*

Abstract

Host-side flash-based caching offers a promising new direction for optimizing access to networked storage. Current work has argued for using host-side flash primarily as a read cache and employing a *write-through* policy which provides the strictest consistency and durability guarantees. However, write-through requires synchronous updates over the network for every write. For write-mostly or write-intensive workloads, it significantly under-utilizes the high-performance flash cache layer. The *write-back* policy, on the other hand, better utilizes the cache for workloads with significant write I/O requirements. However, conventional write-back performs out-of-order eviction of data and unacceptably sacrifices data consistency at the network storage.

We develop and evaluate two consistent write-back caching policies, *ordered* and *journaled*, that are designed to perform increasingly better than write-through. These policies enable new trade-off points across performance, data consistency, and data staleness dimensions. Using benchmark workloads such as PostMark, TPC-C, Filebench, and YCSB we evaluate the new write policies we propose alongside conventional write-through and write-back. We find that ordered write-back performs better than write-through. Additionally, we find that journaled write-back can trade-off staleness for performance, approaching, and in some cases, exceeding conventional write-back performance. Finally, a variant of journaled write-back that utilizes consistency hints from the application can provide straightforward application-level storage consistency, a stricter form of consistency than the transactional consistency provided by write-through.

1 Introduction

DRAM-based caches at the host and at the networked storage system have proven critical in improving storage access performance. Host-side flash is a high-performance and high-capacity caching layer that presents a new opportunity for improving performance when accessing networked storage [6, 12, 13, 23]. Being able to accommodate an order of magnitude more of the working sets of workloads, it creates the potential to significantly improve storage access performance.

Write-through and write-back present two well-known extremes in write policies for host-based flash caches.

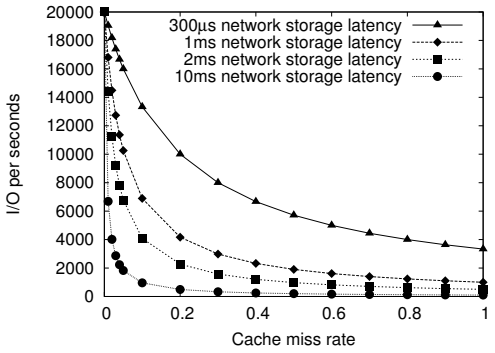

Figure 1: Performance impact of cache misses. *Cache accesses require 50µs.*

Write-through synchronously commits writes to networked storage and then updates the flash cache, before acknowledging the write as completed to the application. This provides the strictest data consistency and durability under all host-level failures thus delivering a *recovery point objective* (RPO) of zero (i.e., zero data loss). Unfortunately, even very small reductions in cache hit rate can result in dramatic reduction in throughput, making the the cache far less effective or even counterproductive for write-intensive workloads. We created a simple model for single-threaded access to storage with a cache to illustrate this. Figure 1 shows that if a cache access requires 50 μs and a network storage access requires 2ms, then the difference in throughput between 99% hit rate and 95% hit rate is about 2x. Thus, when writes are not cached, a potentially large amount of flash throughput could be made unavailable to applications.

Historically, write-back caching has not been considered viable for DRAM caches because they are volatile. Write-back caching becomes attractive with persistent flash-based caches for a variety of performance reasons that we explore in detail in Section 2. Write-back policies, however, have two notable drawbacks. First, since they do not provide strict durability, they introduce data staleness at the networked storage which translates to a non-zero RPO for applications. Second, since conventional write-back can evict dirty data out of order (relative to the original write sequence), it can render the networked storage inconsistent after a host-level failure. Currently available host-side flash caches largely use a write-through policy [6, 12, 17, 23, 1]. When write-

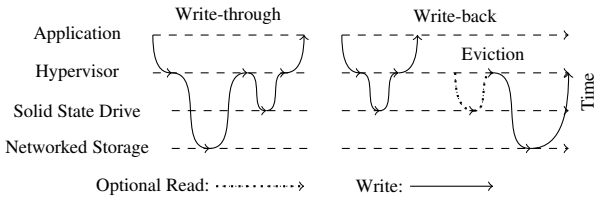

Figure 2: Write-back and write-through policies with a hypervisor managed flash cache.

back implementations exist, these are accompanied with appropriate disclaimers about storage-level inconsistencies after a failure [9, 35]. Network storage inconsistency compromises both data availability after a flash-cache/host failure and the correctness of network storage level solutions such as replication and backup.

In this paper, we develop two new write policies for host-side flash caches that provide close to full write-back performance without compromising network storage consistency. *Ordered* write-back is designed to work seamlessly, requiring no changes to existing storage systems, and outperforms conventional write-through. *Journaled* write-back relies on a logical disk [10] interface that implements atomic write groups at the storage system and offers significant performance gains over the *ordered* policy. Ordered write-back ensures that blocks get evicted from the flash cache and written out to the networked storage in the original write order. Journaled write-back allows overwrites in the cache but ensures that the networked storage atomically transitions from one consistent point to the next. Both policies trade-off strict durability of writes in their design and support an eventually consistent model for the network storage. Under host and/or host-flash failures, these policies do result in data staleness (i.e., loss of recent updates) at the network storage. They thus apply only to applications which can tolerate a non-zero RPO. Journaled write-back additionally allows for a straightforward implementation of *application-level storage consistency*, a stricter form of consistency than the transactional consistency provided by write-through.

We implemented the new ordered and journaled write back as well as conventional write-through and write-back in Linux and evaluated these policies for the PostMark [19], TPC-C [36], Filebench [26], and YCSB [8] benchmarks. The new policies performed significantly better than write-through, with throughput improvements ranging from 10% to 8x for ordered write-back and 50% to 10x for journaled write-back across the four benchmarks. Our sensitivity analysis illustrates the impact of cache size and file system used on the relative performance of these policies. We find that except under extremely low cache sizes, ordered write-back performs better than write-through and that journaled write-back can trade-off staleness for performance, approaching, and in some cases, exceeding conventional write-back performance. Finally, our findings were largely preserved across the four Linux filesystems that we evaluated including ext2, btrfs and three variants each of the journaling ext3 and ext4 file systems.

2 Persistent Write Caches

Persistent host-side flash caches are different than volatile DRAM caches in several respects. In this section, we examine arguments in support of optimizing writes differently in persistent caches and survey the related work in the literature.

2.1 Write Caching

Recent literature has argued in favor of managing persistent host-side flash-based caches as write-through and to optimize exclusively for reads [6, 12, 17, 35]. However, production storage workloads comprise of a significant (often dominant) fraction of writes [5, 21, 29, 32, 37]. Recent studies also report a trend of increasing writes:reads ratios in production workloads [22, 30]. This is a consequence of newer systems absorbing more reads within larger DRAM caches at hosts while all writes get written out to storage for durability. Employing a write-through policy thus, unfortunately, represents a lost opportunity.

Caches that maintain dirty blocks are referred as write-back and caches that do not as writethrough. In case of write-back, the write is acknowledged immediately after the write to the cache. With write-through, writes are first committed to network storage and then to the cache before completion is acknowledged to the guest. Figure 2 illustrates these policies for a hypervisor-managed flash-based cache. As noted earlier, the I/O latency and peak I/O throughput implication of having one policy versus the other is significant (Figure 1). This translates to significantly reduced network storage provisioning requirements with write-back caching for production workloads that are typically bursty [22, 29, 15]

2.2 The Significance of Write-back

A write-back caching policy offers critical performance benefits. First, it significantly lowers write latencies and improves write throughput (per Figure 1); write bursts, if any, get absorbed in the cache, making the best possible use of the high-performance flash layer. Consequently, networked storage can be provisioned for average (instead of peak) write I/O volume. Second, since write-back allows for overwrites (coalescing) in the cache, it reduces the volume of write I/O traffic to the networked storage system for workloads with write locality. Third, since application writes are effectively decoupled from network storage writes, higher levels of I/O parallelism (than available in the application I/O stream) be-

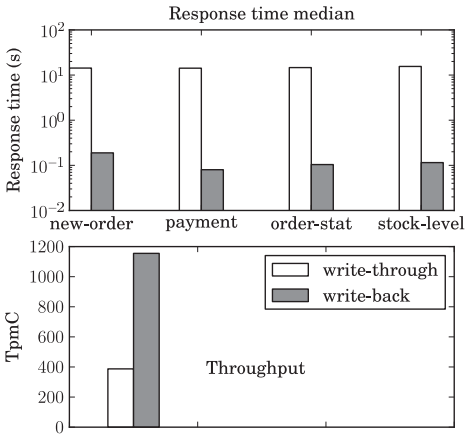

Figure 3: Transaction response times and throughput for TPC-C with 2GB of RAM and 25 warehouses.

	seq write	seq read	rnd read	rnd write
Local SSD	82	93	177	66
iSCSI RAID HDD	891	593	4813	5285

Table 1: Access times in microseconds for local PCI-e SSD vs. networked RAID HDD storage.

come possible when writing to networked storage; storage systems perform better at higher levels of I/O parallelism [4, 7, 11, 16, 33, 34]. Fourth, reads that miss the flash cache experience less I/O contention at the network storage due to write coalescing at the cache layer; the read cache can thus be populated with the changes in the working set more quickly as workload phases change. Finally, since the cache is effective for both reads and writes, cache resizing can potentially serve as a storage QoS control knob (e.g., for I/O latency control) for all workloads including those that are write-intensive.

To quantify the write-back performance advantage, we ran the TPC-C OLTP benchmark which mimics the operations of a typical web retailer including creating and delivering orders, recording payments, checking order status and monitoring inventory levels configured with 10 warehouses [36]. We configured a RAID5 storage array of 8 7200 RPM disks over iSCSI to be the network store and an OCZ PCI-e flash-based SSD as the storage cache on the host. SSD random writes were 80 times faster than the networked iSCSI store. Other aspects of the performance difference are summarized in Table 1.

Figure 3 depicts median response times for the 4 types of TPC-C transactions. Having a write-back cache thus reduces the average response time by at least 75X across the transaction types. Throughput measured as TpmC (new order transactions/minute) showed an increase of 3X with write-back compared to the write-through cache. It is important to point out here that the available SSD

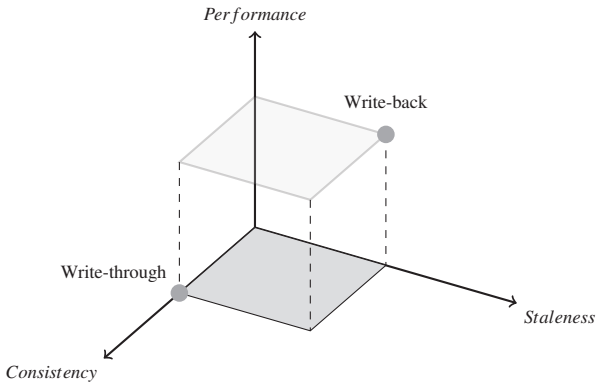

Figure 4: Trade-offs in conventional write caching policies.

throughput was not stressed in write-back mode at the queue depths offered by the workload. For many enterprise applications, lower worst-case performance and lower performance variance are as important as average-case performance. The new write-back caching policies that we develop in this paper target both these metrics by mimicking the basic behavior of synchronous and low-latency local flash updates and asynchronous network storage updates in conventional write-back.

2.3 Performance, Consistency, and Staleness

Write policies make different trade-offs with respect to data consistency, data staleness, and performance as indicated in Figure 4. Conventional write-through and write-back represent merely two extreme points in a spectrum of possible trade-offs. Write-through provides strict durability of writes, i.e., does not introduce any data staleness. On the other hand, write-back fundamentally alters the notion of data durability for applications and introduces both data inconsistency and staleness.

Interestingly, this dilemma has parallels in the remote mirroring for disaster recovery literature. Asynchronous remote mirroring solutions ensure data consistency but introduce data staleness at the target storage system [31, 18, 20, 38]. Key to this argument is the lower cost of data loss after a disaster event when using asynchronous (akin to write-back) mirroring relative to the cost of the high-speed, WAN links necessary to implement fully synchronous (akin to write-through) mirroring to the remote site. While some applications, like financial databases, may require a recovery point objective (RPO) of zero, it has been pointed out that other applications such as non-critical filers and document servers and even online retailers can tolerate non-zero RPO [20]. With such applications, the performance cost of zero data staleness can become prohibitive and trading-off data staleness for performance becomes attractive [20, 31].

Flash-based caches that are backed by slower disk based storage, although representing an entirely different problem domain, present a similar trade-off: the cost of provisioning for peak write bursts at the networked storage to provide strict data durability versus losing some data under host failures due to weak durability. Designing write caching policies that explore the entire spectrum of available trade-offs is key to ensuring that individual applications get to make the trade-offs optimal for their needs.

3 Consistent Write-back Caching

Conventional write-back caching is not employed in a production environment because it can compromise data consistency at the network storage system. In this section, we explore alternate write caching policies that preserve the core advantages of write-back, low latency and write coalescing, while providing a *usable* notion of consistency for the network storage. In designing these new policies, we defined the following goals that allow bridging the performance and consistency gap between the two extremes of write-through and write-back:

- **Goal I**: At the very minimum, ensure *point-in-time* consistency at the network storage so that the network storage always represents a consistent version of the data, albeit at some point-in-time in the past.
- **Goal II**: Turn network storage updates into background operations, thereby providing lower latencies and higher throughput for foreground writes.
- **Goal III**: Preserve and/or increase parallelism of writes to the network storage.
- **Goal IV**: Benefit from write coalescing to efficiently utilize cache space and to reduce write activity at the network storage.

With write-back caches, the network storage receives updates when dirty blocks from the cache are evicted. Key to the two policies that we explore next is controlled eviction to ensure so as to not compromise data consistency at the networked storage. Further, in designing these policies, we trade-off data staleness for achieving a significantly higher level of performance than *write-through*. By ensuring a minimum of *point-in-time* consistency at the network storage, the network storage can be used immediately after a host-level failure that renders the SSD unusable.

3.1 Ordered Write-back

Ordered write-back is based on the simple idea that the original order in which data blocks were updated in the cache can be preserved during eviction. The network storage system is then consistent because it always reflects a valid state as imposed by the storage management layer such as an operating system or hypervisor.

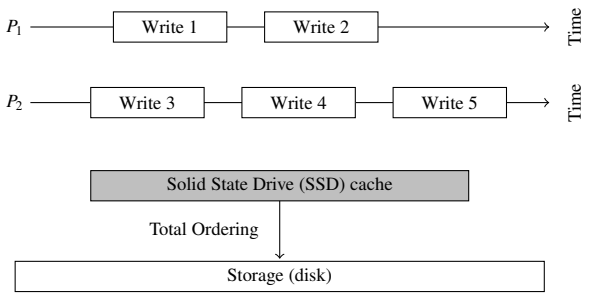

Figure 5: Totally Ordered Write Back.

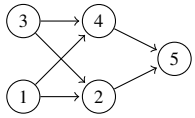

Figure 6: Dependency graph. *Each node represents a write.*

To maintain the original order of writes during eviction, ordered write-back stores the ordering of write I/O operations and the actual data written. Older copies of all data blocks must be preserved in the cache until each of they are written back.

An intuitive approach to store the original order of writes is as a list of blocks sorted by completion time. However, this approach does not allow utilizing parallelism of block writes as available in the original stream. Particularly, writes cannot be sent in parallel even when they are issued in parallel without any dependencies. For example, Figure 5 shows two processes P_1, P_2 issuing some I/Os in parallel. If we used a totally ordered list of blocks we would miss the fact that the I/Os within each of the sets $\{1,3\}$, $\{2,4\}$, and $\{5\}$ can be issued in parallel.

An alternate approach is to utilize actual dependency information between I/O requests. Ideally, an application or file system could provide the cache with accurate dependency information. The current version of our solution developed for the block layer interface is designed to work seamlessly within operating systems or hypervisors. It constructs the dynamic dependency graph online based on the conservative notion of *completion-issue* ordering invariants. A completion-issue ordering invariant requires that if the completion time of block A is earlier than the issue time of block B in the original request stream, then B is dependent on A, otherwise it is not. Following this invariant, a cache can evict all the independent blocks in parallel regardless of the ordering of their completion times.

3.1.1 Write Dependency Graphs

Each node of the dependency graph represents a write I/O operation and contains information about the current location of the block(s) written in the SSD cache as well

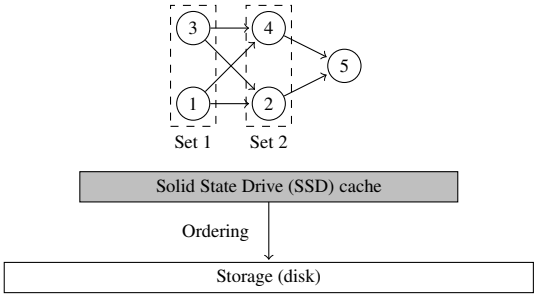

Figure 7: Dependency-induced eviction in ordered write-back. *Eviction of node 5 requires the parallel eviction of nodes in set 1, followed by those in set 2, and then node 5 itself.*

as their permanent location in the network storage system. An edge represents a completion-issue ordering dependency. As an example, let us assume we have the following sequence of I/O issue and completion events: $I_1, I_2, C_1, C_2, I_3, I_4, C_3, C_4, I_5$. Figure 6 shows the related dependency graph. I_i represent an issue event for a write to block i and C_i represents its completion.

To construct and maintain this graph online, nodes are inserted and modified for I/O issue and completion events. The graph is initialized with an empty set $\mathbb{C}$. For an *issue* operation to block X:

1. Add an *incomplete* node X to the set $\mathbb{C}$.
2. Add a link from all *completed* nodes in $\mathbb{C}$ to X.

For a *completion* operation of block X:

1. Mark node X as *completed*.

Notice that we use the observed completions and issuing of the writes to construct the dependency graph. This approach is conservative and may be overestimate dependencies, that is, some dependencies may not be required by the storage management layer. For instance, if 1000 I/Os were meant to be issued independently, we could evict all of them in parallel as well. However, if one of them was completed before another was issued, then we would have to maintain this dependency even though it was not necessary: the application did not wait for the completion of the first write. True dependencies can only be informed by the storage management layer.

Before evicting a dirty block from the cache, the ordered write-back policy ensures that any dirty block(s) that this eviction candidate depends upon and are themselves independent of each other are first evicted in parallel. It performs such dependency-induced evictions recursively until all dirty blocks in the dependency chain are evicted. The evicted nodes are then deleted from the dependency graph. In Figure 7, evicting node 5 would require first evicting the nodes in set 1 in parallel followed by eviction of set 2 before node 5 can be evicted.

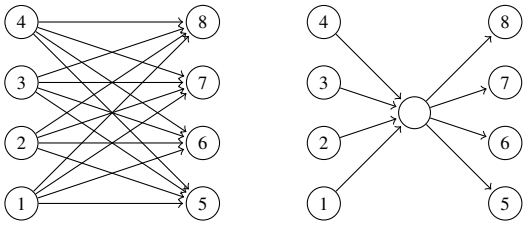

Figure 8: Optimization to reduce the memory requirement for maintaining write dependencies.

3.1.2 Optimizations

Issue and completion operations are computationally inexpensive but memory usage, especially the number of links, can grow up to $\frac{n^2}{4}$ links for a cache size of n blocks. This undesirable property was a significant source of inefficiency in preliminary versions. We found a simple and effective optimization to drastically reduce this link complexity with the use of *dummy nodes*. The optimization inserts dummy nodes after a fixed number of nodes to absorb all completion-issue ordering dependencies due to any nodes representing past write completions. Figure 8 shows the original dependency graph with independent sets $\{1,2,3,4\}$ and $\{5,6,7,8\}$ to the left and the optimized graph on the right hand side with the additional dummy node *D*. Notice how the optimization reduces the number of edges from 16 to 8. Our solution does not explicitly detect high dependency situations such as in Figure 8 but rather simply inserts a dummy node after every 100 I/O completions. Therefore, if the dependency graph does not call for this optimization because of a low link complexity, the optimization introduces a 1% link complexity overhead, but in the best case this optimization can reduce the number of links by orders of magnitude (e.g., 100^2 links to only 200). In practice, this heuristic resulted in excellent memory savings for the workloads we evaluated our system with.

In summary, ordered write-back caching meets a subset of our original goals: I, II, and III. It is unable to coalesce writes in the cache due to the need to preserve the original stream of writes when evicting to network storage. Next, we discuss an improved write caching policy that succeeds in meeting all the four goals.

3.2 Journaled Write-back

Ordered write-back implements consistency by enforcing the same ordering of updates to networked storage during cache eviction as in the original write stream. However, this approach has two drawbacks: first, each block write must be destaged to networked storage and second, all dirty copies of the same block must be maintained in the cache thus wasting precious cache space. *Journaled write-back* addresses both drawbacks by allowing write coalescing in the cache.

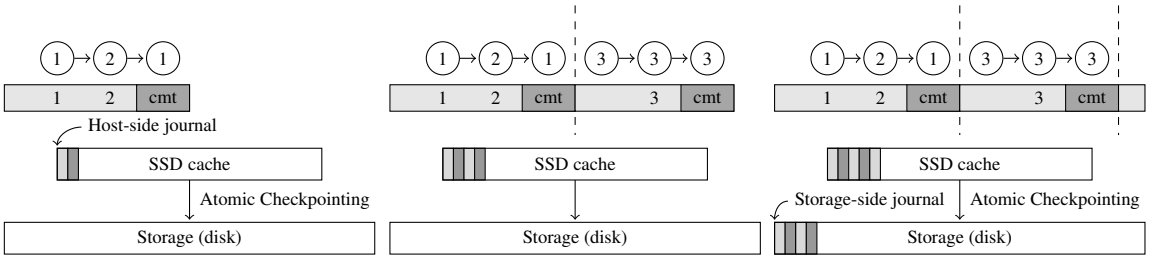

Figure 9: Journaled write-back. *Committing two transactions (left, center) and checkpointing to networked storage (right). The shaded strips represent transaction commit markers that also contain metadata identifying the permanent addresses of various blocks in the transaction.*

3.2.1 Consistency via Journaling

Journaled write-back moves the storage from one consistent state to the next which typically reflects a collection of updates at once instead of a single one. The basic idea is to use grouping of batches of updates in the persistent cache within journal transactions and checkpointing these transactions atomically to networked storage. This is supported by a journal for grouping all updates within the host-side SSD cache within journal transactions; recent updates are always made to the *current* transaction within the host-side journal. Once the current transaction reaches either a pre-defined size or age whichever is met first, it is committed and a new transaction is created and marked as *current*. Commit markers include metadata indicating block addresses in networked storage for individual entries in the transaction to be used at checkpointing time. Transactions are checkpointed to networked storage in the background. Figure 9 illustrates the process of committing and checkpointing two transactions which include updates to $\{1,2\}$ and $\{3\}$. The first transaction includes two updates to block 1 and one to block 2; the second includes three updates to block 3; however, each update is reflected only as a single copy within each committed transaction. These transactions are then atomically checkpointed to networked storage so that networked storage always represents a consistent version of the data at some point-in-time.[1]

The *host-side journal* can have several un-checkpointed transactions allowing transaction commit rate to be independent of transaction checkpointing rate. We found this decoupling extremely useful for supporting write bursts using host-side flash while limiting the amount of data loss under failure modes that do not render the SSD unusable. The networked storage can be provisioned for the average write bandwidth and IOPS usage instead of the peak. Most importantly, this design allows further decoupling of the SSD performance from networked storage performance by allowing write coalescing within individual transactions to reduce the write I/O traffic at networked storage. Finally, since checkpointing is an eager process that frees up cache space, writes do not typically block for cache evictions.

For atomic checkpoints of journal transactions to networked storage, journaled write back implements an *atomic group write* interface, similar to that of the logical disk [10, 14] using an additional journal which we refer to henceforth as the *storage-side journal*. In our implementation, the storage-side journal is an in-memory journal intended for use with NVRAM. Such NVRAM is typically available within medium to high-end networked storage arrays. The NVRAM-based in-memory journal eliminates the need to incur additional storage I/Os for atomic group updates.

3.2.2 Storage Crash Consistency

Checkpointing from the point of view of the host starts with a checkpoint *start* command, a list of blocks and their data to be written, and a checkpoint *end* command. The *start* command is interpreted by the storage by starting a journal entry in NVRAM. Next, the list of block updates are then staged in the journal entry but not made accessible upon reads. Reads to these block addresses prior to the *end* command reception will return the previous versions of the blocks. The *end* command atomically makes the list of block updates available to hosts and marks the entry as checkpointed. The updated blocks are written out to storage in the background; once done, the NVRAM journal entry is deleted. In case of host-level failures that renders the SSD unusable, the host-side journal is simply discarded. The in-memory journal at networked storage ensures that it remains in a stale but consistent state. If the SSD is accessible after the host-level crash, the host-side journal is replayed by checkpointing the un-committed transactions.

3.2.3 Dual Staleness Control

There are two possible states that the system could be in after a host-side crash: *(i)* the host-side cache is not accessible upon recovery either because the host-side SSD has failed or because uptime requirements forces the use of alternate hardware or *(ii)* the host-side cache is accessible upon recovery. If the host cache is not accessible,

[1] Addressing cross-host data dependencies at the network storage (if any) is outside the scope this work.

Policy	Description
WT	*Write-through:* write to disk, write to SSD, and then notify completion to application or guest VM
WB	*Write-back:* write to SSD, on-demand eviction from SSD; SSD-to-disk block mapping stored persistently on SSD
WB-0	*Ordered write-back:* algorithm presented in Section 3.1
WB-J	*Journaled write-back:* algorithm presented in Section 3.2
WB-Jh	*Journaled write-back with application consistency hints:* algorithm presented in Section 3.2 with journal transactions created based on application declared consistency points

Table 2: Write policy descriptions.

Failure	Description
VM or Application	Guest VM or application crashes and will be restarted on the same host
Hypervisor or OS	Hypervisor or bare-metal OS crash and applications will be restarted on the same host
SSD or Host	Hardware failure that renders the SSD and/or host unusable; applications need to be restored on a different host
Network storage	Network storage fails; must be restored from a mirror or from a recent storage-level snapshot/backup

Table 3: Failure modes.

the latest state of the storage is the one in networked storage. If the host cache is accessible, the latest state of the storage is a combination of networked storage and the host-side cache contents. Two distinct staleness outcomes become possible in these two cases. In the former case, staleness is determined by how often the host-side journal transactions are checkpointed: frequent checkpoints leads to lower staleness. Only data within transactions that were checkpointed prior to failure will be available at networked storage. For the latter case, let us assume that transaction commits occur more frequently than checkpoints. In this situation, networked storage staleness after host-side journal replay of un-committed transactions is determined by how often transactions are committed: frequent commits leads to lower staleness. The last few updates to the host would be lost if the last transaction containing these updates was not committed to the host journal.

4 Consistency Analysis

The write policies discussed in the paper differ in the consistency properties they provide. For a single write policy, consistency properties vary based on the presumed failure mode of the system. We start by creating a taxonomy of the write policies and the consistency properties, and discuss the typical failure modes that host-side caches would be subject to. Following this, we address where in the consistency spectrum each of the write policies lie under a specific failure mode.

4.1 Consistency and Failure Modes

Table 2 summarizes the caching policies discussed in this paper. While the first four have been discussed earlier, we introduce a fifth policy, a variant of the journaled write-back that defines block groups to checkpoint aligned with application-defined consistency points to facilitate application-level consistency. This functionality is typical of many enterprise applications that notify operating systems of consistent states (e.g., for volume snapshots/backup). We discuss application-level consistency in more detail later in this section.

Next, we discuss the data consistency properties that are useful to applications. There are several useful notions of consistency that can be afforded to applications and operating systems by a block-level caching layer that interposes on a storage system. *Point-in-time consistency (PiT-C)* ensures that data read after a system crash or failure reflects a consistent version of the storage contents at some point-in-time in the past. Such consistency allows for data staleness due to loss of recent updates. *Transactional consistency (Tx-C)* ensures that data read after a system crash or failure always reflects the most recent update. This is the typical notion of consistency provided by storage systems that are expected to not lose any updates. *Application-level consistency (App-C)* ensures that data read after a system crash reflects a state that is semantically consistent with the application. Application-level consistency relies on hints from the application to define consistency points in storage state. Providing such consistency can enable applications to recover to a state that the application can readily utilize to resume correct execution after a crash. Finally, *application-group consistency (AppG-C)* ensures that data read after a crash or failure is consistent across a group of applications (from the application's standpoint) that are semantically related. Contrasting these policies at a high level, point-in-time consistency is weaker than transactional consistency, while the application-level and application-group consistency are stricter than transactional consistency. However, there is additional subtlety here. When classifying these consistency levels in terms of both strictness and staleness, transactional consistency has no staleness but consistent states defined by it may not make sense from the application's point of view. Its utility depends on the application's ability to make sense of the partial content. Point-in-time consistency has more staleness, but could be functionally equivalent to transactional consistency from the application's point of view since neither pro-

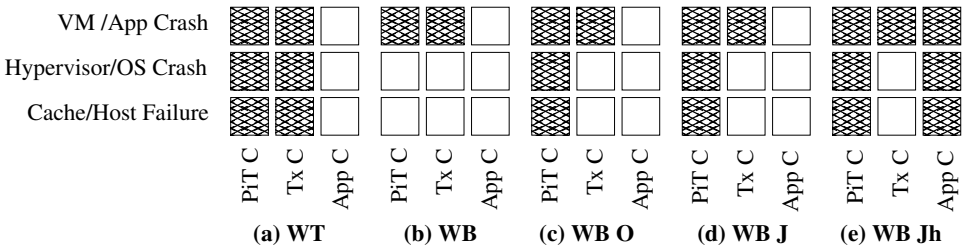

Figure 10: Consistency grid for write policies under different failure modes. *White cells denote consistency property is not met after recovery from crash/failure; patterned cells indicate otherwise.*

vide application-level consistency. Particularly, applications like databases and file systems which are able to recover from storage crashes through careful combinations of logging and write ordering can be restored readily from a point-in-time consistent storage. These applications, however, may not be able to from storage left over from a write-back cache failure since their on-disk log and other data structures will likely be corrupted by the unordered updates. Application-level consistency always makes sense to an application, but it renders storage more stale than transactional consistency and less or more stale than point-in-time consistency.

Consistency properties are relevant, and are typically evaluated, under specific system failure states. Assuming that the host is virtualized, Table 3 lists potential failure modes that the write policies must account for. Of these, addressing network storage failures is beyond the scope of this work. Other non-failure scenarios are also worth examining. For instance, in case of virtualized guests, the VM migration operation can be designed to preserve existing consistency properties. The VM migration operation can be made to flush the SSD cache prior to migration or, alternatively, can be augmented to migrate the state of the SSD cache to the target host's SSD cache, if available. Finally, failure of the VM migration process can be considered equivalent to a hypervisor or OS failure from the standpoint of consistency.

4.2 Consistency with Write Caching

Given the taxonomy above, we now evaluate the consistency guarantees afforded to applications by each of the write policies under each of the failure modes. Consistency is evaluated with respect to the application accessing networked storage via the host. First, no policy provides application-group consistency under any failure mode — achieving such consistency requires collaboration across hosts, which lies outside the scope of this work. Figure 10 provides a consistency grid that maps the remaining consistency properties achievable under the three failure modes by each of the caching policies. Write-through (WT) provides transactional consistency under all failure modes but does not readily provide application-level consistency. Conventional write-back (WB) does not provide any consistency properties under cache/host failure. We assume that, for performance reasons, the WB and WB-O policies do not synchronously update metadata in the SSD cache when data blocks are cached. Thus, neither is able to provide transactional consistency under hypervisor/OS crashes. WB-0 and WB-J both add point-in-time consistency under hypervisor/OS crash and cache/host failure. The journaled write-back policy with application hints about consistency points is the only policy that provides the more powerful and useful application-level consistency property. The fact that it does not provide transactional consistency under many failure modes is not a failing but a feature; after failures, the storage is always left in a state that is consistent with respect to the application. This last application-defined consistent state is likely to be more stale than the transactionally-consistent, current state of storage. Figure 11 summarizes the consistency properties of the caching policies under hypervisor/OS crash and cache/host failures, the key differentiators for the new write policies in contrast to conventional write-back and write-through.

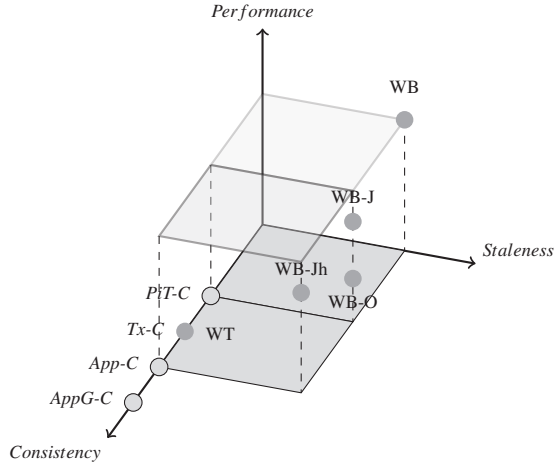

Figure 11: Trade-offs in write caching policies, old and new, under cache/host failures. *PiT-C, App-C, and AppG-C are only reference points along the consistency axis and any write policy implementing these consistency properties will incur non-zero staleness.*

5 Evaluation

The goals of our evaluation are three-fold. First, we evaluate metrics of performance for the new write-back policies, ordered and journaled, and compare them to the conventional write-through and write-back. Second, we study their sensitivity to the major factors that impact the performance of these write policies, including (1) the size of the cache, (2) the size of the journal in the case of write-back journaled, and (3) the file system that was used when running the benchmarks. Lastly, we evaluate the staleness-performance trade-off made possible by the journaled write-back policy by developing cost models for performance and data loss.

5.1 Evaluation Setup and Workloads

Implementation Notes. We implemented a generic cache layer as a module for the Linux kernel 3.0.0 on the host side. This cache implements write-through, write-back and the two new write policies proposed. The cache operates on 4 KB blocks and uses the ARC replacement algorithm [28] that adapts to workloads dynamically and captures both recency and frequency of accesses. For the journaled write-back implementation, we modified the Enterprise iSCSI Target (storage server end-point for iSCSI) [3] implementation. We added an in-memory DRAM-based journal (implemented using NVRAM in practice) and exported a logical disk interface so that the host can specify atomicity of a group of writes during checkpointing.

Testbed setup. The host was running at 2Ghz and configured to run with 512MB to 4GB of memory depending on the working set size of the individual workloads. A 120 GB OCZ REVODRIVE PCI-express SSD attached to the host was used as the host-side cache. The backing store was configured as a RAID 1 array using two 1TB 7.2 RPM disks over iSCSI. The measured performance for these storage devices can be found in Table 1. The ext3 file system was configured to use ordered write-back journaling (unless otherwise mentioned), and the workload was run on top of the bare metal host, i.e., without any virtualization layer. Prior to running each experiment, all memory caches were cleared. We report average numbers across three runs of each experiment.

Workloads. We evaluated the write policies using the PostMark, Filebench-fileserver, TPC-C, and YCSB benchmarks with the ext3 file system (unless otherwise mentioned). PostMark is a file system benchmark that simulates a file server running electronic mail. We configured PostMark to generate a 5:5 mix of 4K reads-vs-appends and creates-vs-deletions over a set of 10000 small files of 100KB each. File buffering was enabled and the machine was configured to run with 512MB of memory. The average working set size for this workload is 2 GB. Filebench-fileserver is a Filebench [26] personality that simulates a server hosting the home directories of a set of users. Each user is simulated by a thread which reads, writes, appends, deletes, creates files on its respective directory. We used the default fileserver configuration with a RAM of size 1GB. The average working set size for this workload is 1.5 GB. TPC-C is an OLTP workload that simulates an online retailer. Virtual users perform delivery and monitoring over product orders. The workload creates multiple threads that perform transactions over a database. We used the TPCC-UVa [24] implementation and configured it to use 50 warehouses and 1GB of RAM. The average working set size for this workload is 9 GB. Finally, YCSB is a framework for comparing the performance of key-value stores. It consists of a multi-threaded client that connects to a given data serving system, MemcacheDB in our case, and generates various types of workloads. We studied how the throughput of MemcacheDB is affected by the different write policies with four of the built-in YCSB workload configurations, A, B, and F, consisting of mix of 50:50 reads:writes, 95:5 reads:writes, and 50:50 reads:writes with read-write-modify behavior respectively. The YCSB client was configured to used 4 threads and records of size 1 KB. The host memory configuration for the YCSB benchmarks was adjusted in proportion to the size of the workload working-sets to introduce adequate I/O activity at the storage system. Finally, unless otherwise mentioned, the journaled write-back experiments used a host-side journal of size 400MB for all the workloads.

5.2 Performance evaluation

In this first experiment, our goal was to quantify the relative performance of the write policies across all the benchmarks. We ran benchmarks against each of the four write policies: write-though, write-back, ordered-write-back, and journaled-write-back, with a fixed cache and journal size. The host-side flash cache size was configured to be close to the working set size for each of the workloads (approximately 80%) so that cache effects can be observed upon workload execution.

Figure 12 presents relative throughput for each of the write policies across all the benchmark workloads normalized based on the throughput with the write-through policy. Write-through performs the worst across all policies. As explained in previous sections, the write-back versions are expected to perform significantly better due to superior write coalescing in the cache and background destageing of dirty data to storage. Further, across most workloads, journaled write-back does better than ordered write-back, approaching and even improving upon conventional write-back performance in some cases. Journaled write-back improves upon ordered write-back by allowing for write coalescing in the cache. In one

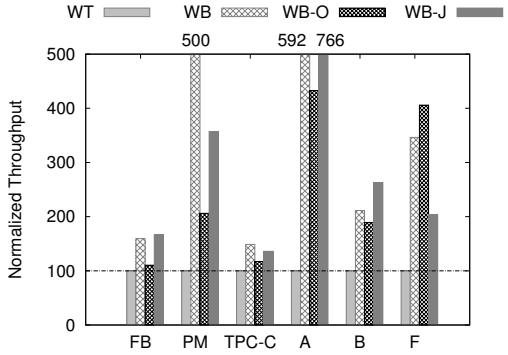

Figure 12: Normalized throughput for Filebench-fileserver (FB), PostMark (PM), TPC-C, and YCSB workloads-A, B, and F.

case (YCSB-F), ordered write-back performs the best, even surpassing both conventional and journaled write-back variants. This occurs due to two reasons: *(i)* unlike conventional write-back, ordered write-back benefits from dependency-induced batching of evictions from the cache, and *(ii)* journaled write-back was configured with a small journal in this experiment relative to the workload's working-set leading to too many evictions relative to ordered and conventional write-back. Finally, journaled write-back can sometimes perform better than conventional write-back because it benefits from eager, grouped checkpointing for destageing dirty data whereas conventional write-back destages dirty data only on demand. Later in this section, we demonstrate how the performance with the journaled write-back policy can be tuned to meet write-back performance depending on application staleness tolerance.

5.3 Sensitivity Analysis

The performance of the write policies are sensitive to the size of the cache, the size of the journal, and the differential shaping of application I/O traffic induced by different file systems. We used the Filebench-fileserver and PostMark workloads to better understand the performance sensitivity of the write policies to these factors.

5.3.1 Sensitivity to Cache Size

We ran Filebench-fileserver using each of the write policies under several cache sizes. Journaled write-back was configured to limit the journal size, and as a result, the maximum staleness at the network storage to 100 MB. Figure 13(a) shows fileserver throughput performance in operations per second. Filebench-fileserver is a write intensive workload and therefore insensitive to cache size in the case of reads. We see that that write-through performance is similar across all cache sizes since all writes must be written synchronously to network storage. The other policies have two dominant trends. For cache sizes greater than 1.5GB, we observe the expected

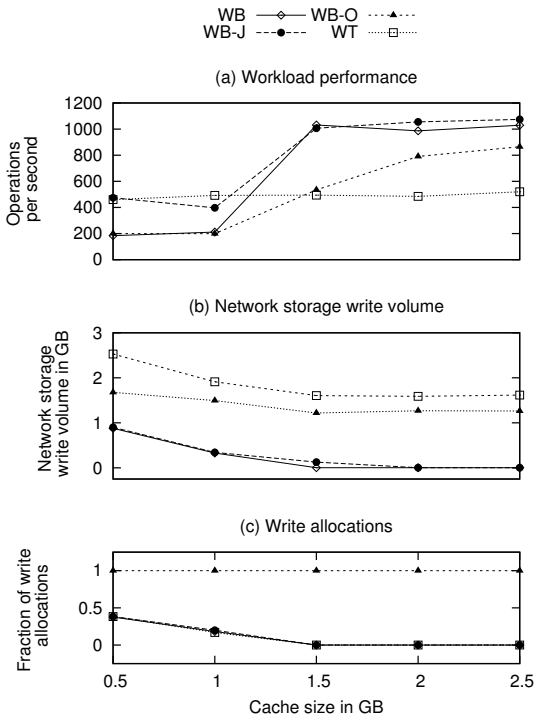

Figure 13: Filebench-fileserver performance for all four write policies at different cache sizes.

trend across the write policies. All three write-back policies perform better than write-through. Write-through, as shown in Figure 13(b), has the highest write volume to network storage. The write-back policies coalesce writes, reducing the write volume to network storage. Among the write-back variants, ordered write-back has lower relative performance, 800 compared to 1000 operations per second for conventional write-back. A *write-allocation* indicates an operation whereby a write requires additional space in the cache. Since all writes in case of ordered write-back need to be made to a new location to avoid overwriting previous versions of the block, they all require write-allocation (Figure 13(c)). Write-allocations can be expensive; if eviction is necessary to allocate and if the block selected to be evicted is dirty, a write to network storage becomes necessary. This affects the performance of ordered write-back negatively the effects of which can also be observed in Figure 13(b): more write traffic to network storage in the case of ordered write-back.

The second trend occurs at cache sizes less than 1.5GB of cache size where write-through and journaled write-back perform significantly better than the other two policies. For small cache sizes, most cache writes induce write-allocation and the resulting dirty block evictions from the cache in case of the write-back policy variants dominate cache behavior which negatively affects performance with these policies. Interestingly, write-through

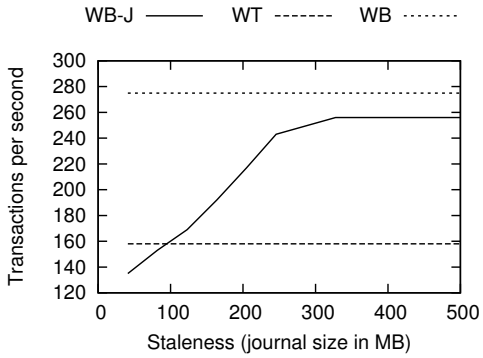

Figure 14: PostMark performance with 1.5GB of SSD cache for varying staleness set as the maximum number of dirty pages.

does well since all write-allocations in write-through are "free"; they do not induce additional writes to network storage as all the cached blocks are always clean. In the case of journaled write-back, the write-allocations are relatively less expensive than the other write-back variants because the eviction of dirty data is batched and therefore more efficiently written to network storage. Next, we evaluate the sensitivity of journaled write-back policy to the host-side journal size.

5.3.2 Sensitivity of WB-J to Journal Size

Journaled write-back does not need to store copies of old blocks and therefore has minimal overhead relative to conventional write-back. The reason why it performs worse than write-back in some of the previous experiments is that it limits the amount of staleness by restricting the size of the host-side journal. Reducing the journal size increases the probability of cache evictions during transaction commits. To evaluate the sensitivity of journaled write-back performance to the allowable staleness of storage, we conducted an experiment where we fix the cache size to 1.5GB and vary the allowable staleness (host-side journal size). Figure 14 depicts how performance can be tuned to span a significant portion of the range between write-through and write-back by varying the staleness tolerance for the PostMark benchmark. A larger journal allows for greater write coalescing and batching of write traffic to the network storage and thus aids performance, but it also results in greater staleness at the network storage after a host-level failure. The journal size is an ideal knob to achieve an application-defined performance/staleness trade-off using the journaled write-back policy.

5.3.3 Sensitivity to the File System Type

File systems alter write ordering and impose additional synchronous write requirements. In this regard, file system designs vary significantly. The host-side cache operates at the block layer (either within the OS or hypervisor) and alters the I/O stream created by the file system operating at some layer above it. It is therefore important to evaluate how the performance afforded to applications by individual filesystems are impacted due to this additional layer and policies within it.

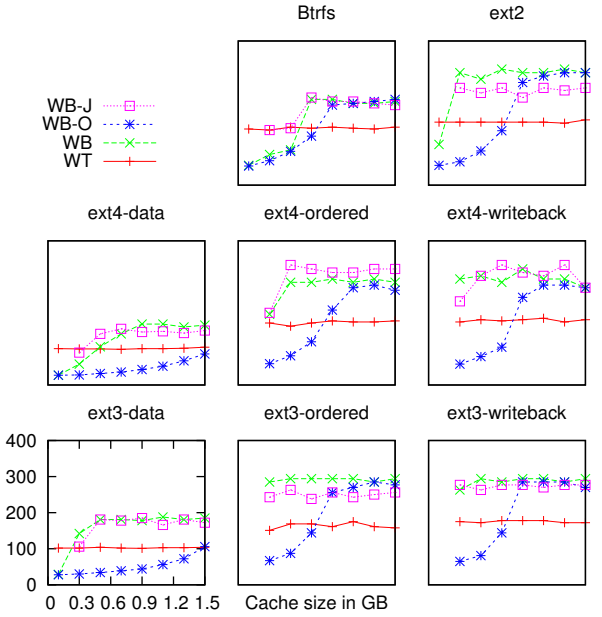

Figure 15: PostMark transactions per second under different file systems. *The axis ranges are indicated in the bottom-left plot and are the same across all the sub-plots. Notice that ext*-writeback and ext*-ordered are referring to a journaling mode and not to the cache's write policy.*

We studied 4 different file systems, ext2, ext3, ext4, and btrfs. Ext2 is a file system based on the traditional Berkeley FFS [27]. Ext3 implements a journaling layer on top of ext2 whereby metadata (and optionally, data) writes are initially directed to a journal, and later checkpointed to their final location within the ext2 on-disk structure. We evaluated all three journaling modes of ext3: *writeback* and *ordered*, where only the metadata is journaled, and *data*, where both data and metadata are journaled. Figure 15 depicts performance for the PostMark benchmark for all write policies as we change the underlying file system at various sizes of the host-side SSD cache. This set of experiments revealed a set of interesting insights about how these file system designs are impacted by an SSD caching layer operating below them. We discuss these next.

The first broad trend we draw is that across all the file system variants, we notice that the write-back caching policy variants outperform the write-through policy for cache sizes that are sufficiently large so as to accommodate a sizable fraction of the workload's working set size (2GB in this case). Second, the ordered write-back caching policy provides performance superior to write-through beyond a certain cache size for all file systems,

and that this improvement is larger for journaling file systems. Journaling file systems introduce additional data writes and this increases the size of the write dependency graph. Thus, ordered write-back is worse than write-through at small cache sizes; it maintains multiple copies of the dirty blocks in the cache thereby affecting its cache hit ratio. Third, we also notice that since data mode journaling introduces significant additional writes to storage, it incurs a significant performance penalty relative to ext2 and the other journaling modes for both ext3 and ext4.

Considering ext3 relative to ext2 and with write-back caching, for a cache size of 1.5 GB, the ext2 obtains the highest throughput of 300 transactions-per-second in contrast to 190 for ext3-data and 290 for ext3-writeback. The amount of metadata operations is not significant and thus the performance of ext3-writeback and ext3-ordered are not significantly impacted.

Ext4 is an extension of ext3 designed to address the limits placed by ext3's data structures (e.g. file system size and file sizes) as well as to improve ext3's performance further [2]. The most relevant of the new features in ext4 are delayed and multi-block allocation which improve the contiguous allocation of disk blocks. Given that SSDs are designed for random accesses, large contiguous allocations are not expected to significantly benefit workloads cached in the SSD. Looking at Figure 15, we observe that for cache sizes above 0.6GB, the write-back policy performs similarly in ext3 and ext4. Additionally, write-through performs largely similarly in ext3 and ext4 for all journaling modes. An outlier here is ext4-ordered, where the journaled write-back performs better than regular write-back. We believe this is due to the contiguous allocation feature in ext4. Journaled write-back checkpoints transactions, batching the appends of a large number of files to network storage. By using ext4 at the disk-based network storage, we increase the chances of aggregating sequentially writes to disk and therefore increasing overall performance. Evictions in conventional write-back are unable take advantage of this behavior because these evictions do not occur in batches.

The last file system we studied is *btrfs* which is a copy-on-write file system designed for scalability and reliability [25]. Btrfs has a similar behavior to the extended file systems for large cache sizes: the write-back policies perform better than write-through. An interesting observation here is that write-through performs similarly with btrfs and ext2 but the gap between write-through and the write-back policies is about 50% smaller in case of btrfs. In other experiments (not shown), we found that the PostMark workload performs similarly with disk drives when using btrfs and ext2, but worse with SSDs when using btrfs compared to ext2. We did not use any of the SSD optimizations offered by the more recent versions of btrfs in any of our experiments.

5.4 Trading Staleness for Performance

The work of Keeton *et al.* [20] explored several disaster-tolerant storage solutions and demonstrated that a trade-off can be made evaluating the solution implementation costs (including both the necessary storage and network facilities) and the cost of data loss after a disaster. It argued that for an application that can tolerate non-zero RPO (e.g., a filer storing non-critical documents; a web server where data can be replaced from other sources; or social-network data that can tolerate negligibly probabilistic losses), the optimal storage solution is not necessarily one that provides the strongest durability (e.g., synchronous remote mirroring with a network link sufficiently provisioned for the peak write bandwidth) simply because the implementation cost often far exceeds the cost of a small amount of data loss. A similar argument applies when evaluating host-side write caching policies.

For an application requiring a RPO of zero, write-through must be employed but the network link and back-end storage must also have sufficient bandwidth to satisfy the short-term burst write rate. For other applications, write-back caching can be used to lower the cost of the network and storage hardware. The hardware would need to support only the long-term average write rate if using ordered write-back, or the rate of unique writes in a write-back window if using journaled write-back. Hence, the optimal write policy can be determined by considering the cost of potential data loss and the cost of network and storage hardware necessary to support a certain peak data access rate. While both the ordered and journaled write-back policies support such a trade-off, we now present a case-study with the journaled write-back policy which provides a straightforward knob (the journal size) to trade-off different amounts of staleness for performance.

For the journaled write-back policy, we note that a quantitative cost-benefit analysis can be performed to determine the optimal write-back journal size, w, by trading data loss and associated cost against the application performance generated revenue. The former can be determined through an offline profiling process that maps w to the application performance; the latter can be specified by the application user or system administrator (e.g., [20]). Ultimately, both can be expressed in terms of dollars as part of the service-level agreement (SLA), so the problem becomes finding the optimal write-back journal size w^* as follows: $w^* = argmax_w(Revenue(w) - DataLossPenalty(w))$.

To illustrate this cost-benefit analysis, we consider an example using the Postmark benchmark to mimic the operations of a news server storing non-critical news items where a limited amount of data staleness is tolerable after server (host) failure. For the revenue model, we assume that the revenue generated from this application

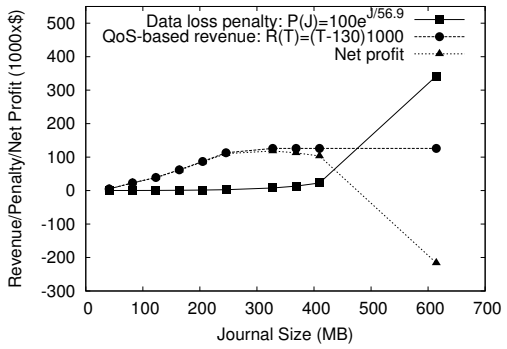

Figure 16: Trade-off between application QoS-based revenue and data loss penalty as the journal size used with journaled write-back caching is varied.

is a linear function of benchmark throughput to reasonably reflect a fixed $ revenue generated per transaction per second. Based on the performance-staleness profile of this application (Figure 14), we can then derive its revenue as a function of the journal size. For the data loss penalty model, we assume that the data loss penalty is an exponential function of the data staleness (journal size) to reflect the exponential cost of losing customer base due to an inadequately consistent service. Further, the function accounts for probability of system crashes and failures that lead to data loss. Both the revenue and data loss penalty models are illustrated in Figure 16 as functions of the journal size. The difference between these two functions, i.e., the net profit generated from the service, is also shown; the optimal journal size is around 650MB. Although the exact shapes of the revenue and penalty functions used here are assumed rather arbitrarily, they can be determined in practice based on the SLA specifications and the combined probability of data loss under various failure modes. This example clearly demonstrates how a more profitable trade-off can be made between performance-generated revenue and data loss penalty after a failure by finding the optimal journal size.

6 Conclusion

For decades, write caching policies have been monopolized by the two extremes of transactionally consistent but low-performing write-through and high-performing but inconsistent write-back. However, the spectrum of possible write policies is significantly richer, offering a variety of trade-offs across performance, consistency, and staleness dimensions. Exploring this spectrum is even more relevant now with the availability of high-performance, persistent host-side caches. In this paper, we develop new write caching policies that strike new and useful trade-offs in this spectrum. Ordered write-back provides point-in-time consistency with significant improvement in performance over conventional, transactionally consistent, write-through. Journaled write-back relies on an extended, but straightforward storage interface modification to provide point-in-time consistency and superior performance than the ordered mode. Journaled write-back also provides a simple knob to trade-off data staleness for performance that allows it to achieve close to write-back performance. Further, a variant of journaled write-back that uses application level consistency hints can provide application-level consistency, a stronger type of consistency than the transactional consistency that conventional write-through provides. The flexibility afforded by these new write policies enable write caching mechanisms that are better tailored to the needs of individual applications.

Acknowledgments

The authors thank the anonymous reviewers and shepherd, Xaiosong Ma, for their thoughtful feedback. Irfan Ahmad, James Lentini, and Jiri Schindler provided valuable input as well. This work was supported by NSF grants CNS-1018262 and CNS-0747038 and a NetApp Faculty Fellowship.

References

[1] Bcache. http://bcache.evilpiepirate.org.

[2] Ext4 (and Ext2/Ext3) Wiki. http://ext4.wiki.kernel.org.

[3] IET: The iSCSI Enterprise Target Project. http://iscsitarget.sourceforge.net.

[4] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design Tradeoffs for SSD Performance. In *Proc. of Usenix ATC*, Boston, MA, June 2008.

[5] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis. BORG: Block-reORGanization and Self-optimization in Storage Systems. *Proc. of USENIX FAST*, February 2009.

[6] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer. Mercury: Host-side flash caching for the data center. In *Proc. of IEEE MSST*, April 2012.

[7] P. M. Chen and E. K. Lee. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2), June 1994.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *Proc. of ACM SoCC*, 2010.

[9] J. Corbett. A bcache update. http://lwn.net/Articles/497024/.

[10] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A new approach to improving file systems. In *Proc. of ACM SOSP*, 1993.

[11] C. Dirik and B. Jacob. The performance of pc solid-state disks (ssds) as a function of bandwidth, concurrency, device architecture, and system organization. In *Proc. of ISCA*, 2009.

[12] EMC. VFCache. http://www.emc.com/storage/vfcache/ vfcache.htm, 2012.

[13] Fusion-IO. ioTurbine. http://www.fusionio.com/systems/ioturbine/, 2012.

[14] R. Grimm, W. C. Hsieh, M. F. Kaashoek, and W. de Jonge. Atomic recovery units: Failure atomicity for logical disks. In *Proceedings of the IEEE International Conference on Distributed Computing Systems*, 1996.

[15] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proc. of USENIX FAST*, February 2011.

[16] A. Gulati, I. Ahmad, and C. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proc. of USENIX FAST*, February 2009.

[17] E. V. Hensbergen and M. Zhao. Dynamic policy disk caching for storage networking. Technical Report Report (RC24123), IBM Research, November 2006.

[18] M. Ji, A. Veitch, and J. Wilkes. Seneca: Remote mirroring done write. In *Proc. of USENIX Annual Technical Conference*, June 2003.

[19] J. Katcher. PostMark: A New File System Benchmark. *Technical Report TR3022. Network Appliance Inc.*, October 1997.

[20] K. Keeton, C. Santos, D. Beyer, J. Chase, and J. Wilkes. Designing for disasters. In *Proc. of USENIX FAST*, 2004.

[21] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proc. of USENIX FAST*, pages 211–224, February 2010.

[22] A. Leung, S. Pasupathy, G. Goodson, and E. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of USENIX ATC*, 2008.

[23] A. Leventhal. Flash storage memory. *Commun. ACM*, 51:47–51, July 2008.

[24] D. R. Llanos. Tpcc-uva: An open-source tpc-c implementation for global performance measurement of computer systems. *ACM SIGMOD Record*, December 2006. ISSN 0163-5808.

[25] C. Mason. The btrfs filesystem. *The Orcale cooperation*, 2007.

[26] R. McDougall. Filebench:application level file system benchmark.

[27] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, Aug. 1984.

[28] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proc. of USENIX FAST*, pages 115–130, 2003.

[29] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. *Proc. of the USENIX FAST*, Feb 2008.

[30] D. Narayanan, A. Donnelly, and A. Rowstron. Write Off-Loading: Practical Power Management for Enterprise Storage. In *Proc. of USENIX FAST*, 2008.

[31] H. Patterson, S. Manley, M. Federwisch, D. Hitz, S. Kleiman, and S. Owara. Snapmirror: File system based asynchronous mirroring for disaster recovery. In *Proc. of USENIX FAST*, January 2002.

[32] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of USENIX ATC*, 2000.

[33] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 2:17–28, 1994.

[34] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proc. Winter 1990 USENIX Technical Conference*, 1990.

[35] M. Srinivasan. Flashcache : A Write Back Block Cache for Linux. https://github.com/facebook/flashcache/blob/master/doc/flashcache-doc.txt.

[36] Transaction Processing Performance Council (TPC). TPC Benchmarks. http://www.tpc.org/information/benchmarks.asp.

[37] A. Verma, R. Koller, L. Useche, and R. Rangaswami. SRCMap: Energy Proportional Storage using Dynamic Consolidation. In *Proc. of USENIX FAST*, pages 267–280, February 2010.

[38] H. Weatherspoon, L. Ganesh, T. Marian, M. Balakrishnan, and K. Birman. Smoke and mirrors: Reflecting files at a geographically remote location without loss of performance. In *Proc. of USENIX FAST*, 2009.