Exploring Caching Policies for Host-side Flash Caches

Bhavesh Singh
143059003

Seminar Report
submitted in partial fulfillment of the
requirements for the degree of
Master of Technology

under the guidance of
Prof. Purushottam Kulkarni

Department of Computer Science and Engineering
Indian Institute of Technology Bombay
India – 400076

April 2016

Contents

List of Figures

List of Tables

Abstract

Caching is a widely employed technique in computer systems to overcome the disparity in the speeds of the processing and memory subsystems. It is possible to mostly serve data out of a faster, but smaller sized memory unit because of the principle of locality of reference (both spatial and temporal). Caches are used at different levels in the memory hierarchy. For example, processor caches store recently accessed data from primary memory (RAM), disk caches are used to store recently accessed disk blocks, etc. In data centers the storage is typically separated from the compute nodes and is accessed over the network. Solid state disks (SSDs) have been employed in such data centers as caches on the compute nodes. Such caches are called host-side caches. Recent research [7, 4, 8] has focused on improving the performance of these caches for write-mostly scenarios. The common approach is to use some variant of write back to send the writes asynchronously to the networked storage. But such policies risk data loss in case of failure. In this seminar we look at the various caching policies for host-side SSD caches and explore the possibility of providing probabilistic guarantees to asynchronous writes, thus trying to minimize loss of data in the case of failure. We try to define the problem by looking at various components and suggest possible solution approaches.

1 Introduction

Data centers typically have a centralized storage architecture. *Centralized storage* architecture consists of a *storage area network* (SAN) storage with multiple hosts reading/writing data. The SAN storage exposes a number of logical disks which can be mounted by the hosts. Figure 1 shows the typical centralized storage architecture with multiple application servers talking to a single SAN storage over the network.

There are various advantages of using this architecture (which mainly come from the physical separation of the storage from the servers) — storage can be scaled as required, maintenance becomes easier and backups are managed easily. Moreover, the total storage capacity utilization increases due to the consolidation of data from many servers.

But accessing storage across the network is slower than accessing a direct attached storage. Moreover, the centralized storage could potentially become a bottleneck if there are a lot of writes from the hosts. Because of this, caching solutions have been sought after that can save round trips to the networked storage.

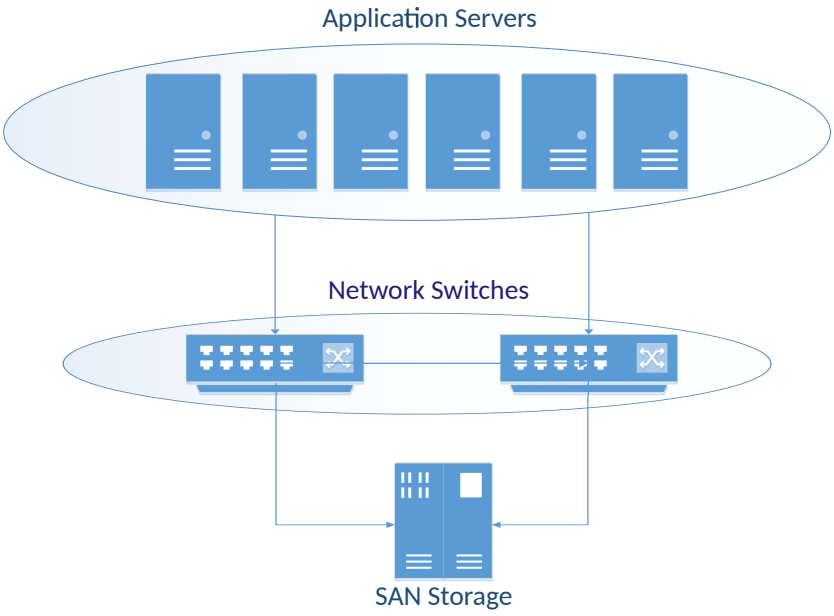

Figure 1: Centralized Storage (adapted from [11])

A *solid state disk* (SSD) is a flash based device that is increasingly being used as a persistent disk cache on the host. These caches, aptly called host-side flash caches in the literature, help save a network

round-trip to the storage server in case of a cache hit. This helps alleviate the problems of bottlenecks at the storage server to a large extent. Flash based device access speeds are slower than DRAM but faster than spinning disks and SAN storage. Figure 2 shows the SSD in the memory hierarchy.

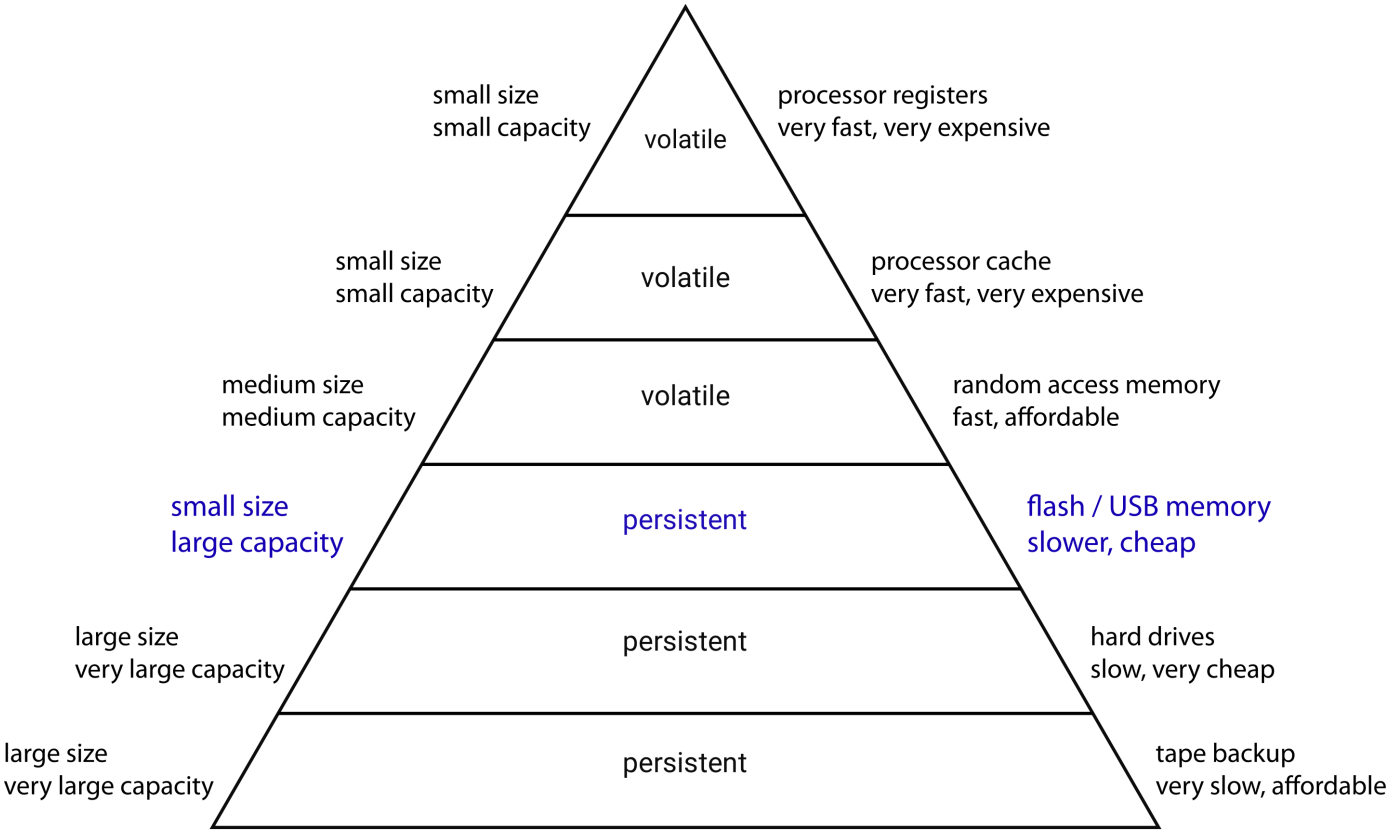

Figure 2: Memory hierarchy (taken from [10])

The employment of a host-side cache has challenges of its own. An appropriate caching policy needs to be employed (see Section 2). The traditional write through might increase write latency, while write back might cause consistency issues. Current research (see Section 3) focuses on using host-side caches as write caches while maintaining consistency. Interestingly, remote mirroring across wide area network (see Section 4) has parallels in the caching trade off of write speed vs consistency. In this seminar, we look at the various caching policies and try to motivate the need for another probabilistic policy that can provider better performance and consistency guarantees (Section 5).

2 Caching Policies for Host-side SSD Caches

Host-side flash caches can speed up reads on the host by saving network round-trips to the storage server. We discuss two well known caching policies in the context of host-side caches in this section. But before that we define data consistency which is central to our analysis of caching policies throughout this report.

2.1 Data Consistency

Data consistency refers to the parity between the storage (in our case, networked storage) and the view of the storage by the application. There are different definitions of consistency and they reflect different states of the storage with respect to the application's view. *Transactional Consistency* means that the storage reflects the most recent update done by the application at any point in time. Here update means the writes that were acknowledged as successful to the application. *Point-in-time Consistency* ensures that after recovery from a failure, the storage state will reflect a state that matches the application's view of

the storage at some point in time in the past. *Application-level Consistency* is a stronger guarantee that ensures that after a failure, the storage state will match some constraints according to the semantics of the application. This is more useful to the application because it can start working immediately without needing recovery.

2.2 Write Through Policy

Figure 3 illustrates the write through policy. The broad steps are as follows –

(1) Writes are synchronously done on the networked storage.

(2) Cache is updated.

(3) The write is acknowledged to the application.

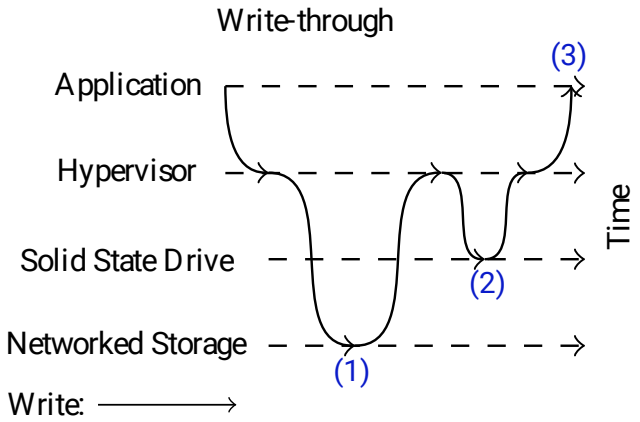

Figure 3: Write Through Policy (taken from [7])

Advantages The biggest advantage of write through is that it guarantees transactional consistency. So all writes are guaranteed to be persisted on the networked storage and no data is lost in case of failure.

Disadvantages The policy suffers from low write throughput. The network storage is accessed on each write creating a bottleneck at the server for write heavy workloads across multiple hosts. The network bandwidth is also an issue between storage server and hosts.

2.3 Write Back Policy

The write back policy (figure 4) is on the other extreme. The broad steps are as follows –

(1) Writes are written to flash cache.

(2) Immediate acknowledgment is given to the application.

(3) Updates sent to the networked storage whenever there is an eviction of a dirty page. Eviction can happen in two ways — eviction on memory pressure using cache eviction policy, or a daemon which can evict data in the background continuously.

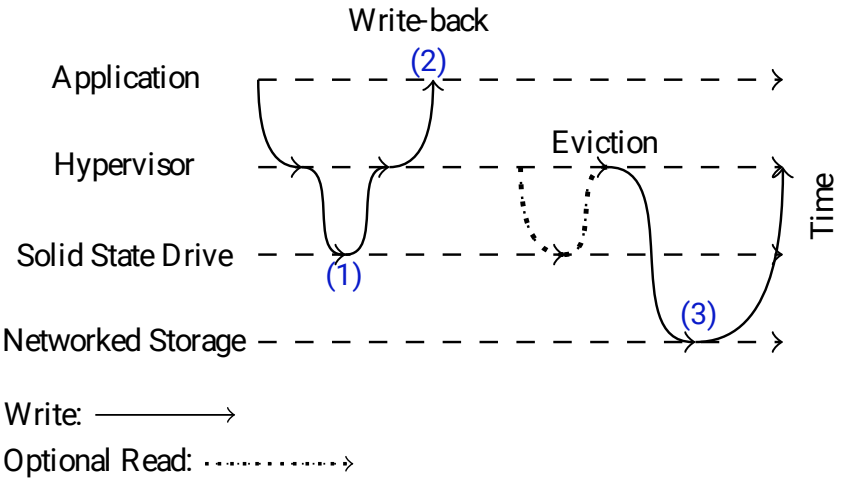

Figure 4: Write Back Policy (taken from [7])

Advantages Write back can aid in achieving high I/O throughput. The writes are acknowledged as soon as they are done locally. The possibility of write coalescing results in low usage of network bandwidth to send writes to networked storage each time. This also results in low load on networked storage server. Parallel eviction of writes is possible with write back, if needed. Although the traditional policy evicts blocks only on memory pressure and if it is dirty, it is written back to the storage server.

Disadvantages There is no consistency guarantee at all. If flash failure occurs, data is bound to be lost and leave the networked storage in an inconsistent state. There is no way to recover from such a state. If only the host fails and the flash device is intact and there is enough metadata on the flash device to read and send all updates from it to the networked storage, only then will this policy work. But this usually translates to variable recovery time.

3 SSD Caches as Write Caches

Traditional usage of SSD caches on host side favored the write through policy. This is beneficial only when the workloads are read-heavy. The host side cache is not able to provide any benefits in the case of write-heavy workloads. The write through policy ensures that all writes are done synchronously to the networked storage. Koller et al. [7] propose the idea of using host side SSD cache for caching the writes too on the host-side cache instead of sending them to the networked storage, just like write back. But write back provides no consistency guarantees at all in the case of failures.

3.1 Variants of Write Back Policy

Koller et al. [7] propose two variants of the write back policy — **ordered write back** and **journaled write back**. These variants, while trying to take advantage of write back like semantics (i.e. acknowledging a write as soon as it is written to the flash cache), also maintain consistency. Specifically, these write policies provide *point-in-time consistency* as the writes are not immediately reflected to the networked storage but stored in the cache.

Qin et al. [8] propose other important variants of write back policy — **write back flush** and **write back persist**. These variants provide the stronger consistency guarantee — *application-level consistency*.

All of the variants of write back policy that we study here follow the steps outlined in figure 5.

(1) The application does a write.

(2) The write is acknowledged after persisting the data on the flash cache.

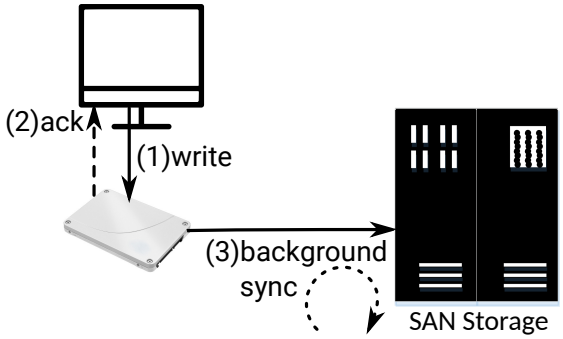

Figure 5: Write Back Variants

(3) A background process is responsible for evicting the updates to the networked storage.

It is in step (3) that most of the difference lies in various policies. For example, journaled write back evicts writes in batches called journals.

3.1.1 Ordered Write Back

The ordered write back policy ensures the eviction of writes to blocks in the same order in which they were written to the flash device. This ordering ensures that in case of failure, the networked storage is point-in-time consistent. This ordering is implemented by maintaining an in-memory dependency graph of block writes. The graph reflects the *completion-issue* partial order. If block 2's write is issued after block 1's write completion, then block 2 is dependent on block 1. This means that block 1 will have to be evicted before 2 is evicted. Figure 6 shows an example of such dependency graph. It shows two independent sets whose blocks can be evicted in parallel without loss of consistency.

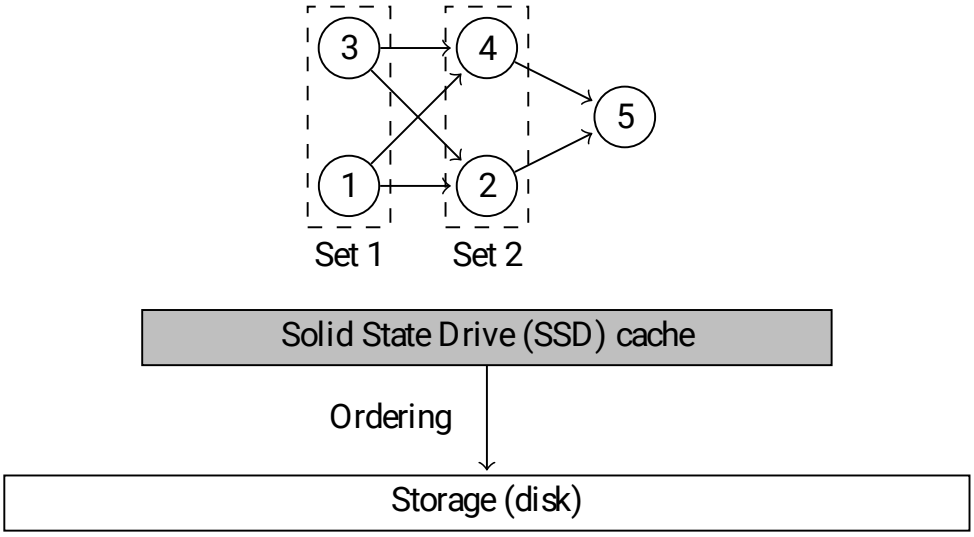

Figure 6: Ordered Write Back (taken from [7])

Advantages Ordered write back may increase the chances of parallel writes to the networked storage as the writes are being cached and not immediately written like write through. It can be implemented easily without any modifications to the interface of the networked storage.

Disadvantages Ordered write back needs each block write to be stored in the flash layer separately. This means that multiple writes to the same block but separated by other writes also go to different locations. Thus, the cache can get filled quickly due to absence of write coalescing (the ability to overwrite a copy of

the block with an updated version) and hence the eviction rate of writes has to be faster. Ordered write back, thus, runs the risk of overwhelming the networked storage with a lot of writes.

3.1.2 Journaled Write Back

Journaled write back is an extension of ordered write back and relies on the networked storage supporting *atomic group writes*. This means that the storage should reflect the updates to a group of blocks atomically – either all the block updates are seen or none. Based on this premise, this policy maintains journals in the host-side cache. Block writes are logged in this journal. If a block already present in the journal is updated, its entry is overwritten i.e. write coalescing is done. When a journal reaches a fixed size (number of blocks), it is *committed* and a new journal is started. A daemon is responsible for *checkpointing* the journal to the networked storage, which, as mentioned earlier, is an atomic process. The correctness of this policy is ensured by the atomic group updates. This is because the *state of the storage goes from one point-in-time consistent state to another*.

Figure 7 shows the process of journal commit and checkpointing. Note that the journal contains only one entry for block 3 even though multiple writes are done to it. Thus write coalescing helps save cache space.

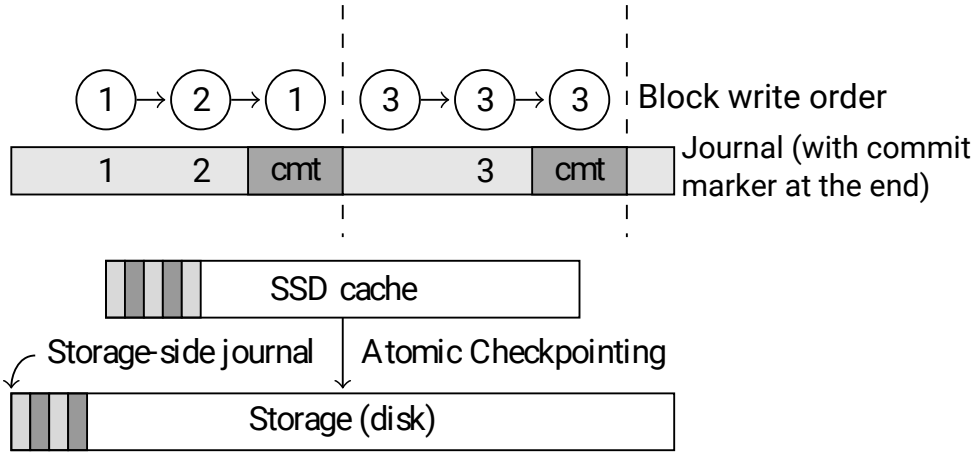

Figure 7: Journaled Write Back (taken from [7])

Advantages In addition to the advantages provided by ordered write back like point-in-time consistency, improved write speeds for the application, parallel eviction of writes, journaled write back also saves cache space by coalescing writes within the same journal.

Disadvantages Journaled write back needs modifications to the networked storage interface to support *atomic group writes*.

3.1.3 Write Back Flush and Write Back Persist

Write back flush and Write back persist follow the philosophy that a cache should only provide the same reliability guarantees as the underlying storage, which is that data is guaranteed to be persisted at every write barrier. This enables these policies to ensure that the networked storage is application-level consistent.

The application-level consistency guarantee is achieved by looking at the `fsync` system call which is equivalent to a write barrier. All the dirty blocks in the cache are flushed to the networked storage at each barrier. This ensures that at any point of time, the storage represents a state in which the application was consistent. Since the applications do not expect the data to be persistent between barriers, the caching policies are not bound to support it.

The only difference between write back flush and write back persist is in the mechanism of flush involved. In write back persist, instead of evicting all dirty blocks, the metadata about all the dirty blocks is persisted on the flash cache itself. The eviction is done in the background. This helps achieve better performance of writes than write back flush but can lead to loss of data in the case of flash failure.

3.2 Fault-tolerant Write Back Policy

Bhagwat et al. [4] combine ordered write back with synchronous replication to `k` other peers — **write back with peering**. This policy includes an additional step of replicating each write synchronously to the flash caches of `k` 'peer' hosts. The number of hosts is configurable with parameters like same or different failure domains etc. Figure 8 shows the steps followed.

(1) The application does a write.

(2) The update is synchronously replicated in the flash caches of `k` peers.

(3) The write is acknowledged to the application.

(4) A background process is responsible for evicting the updates to the networked storage.

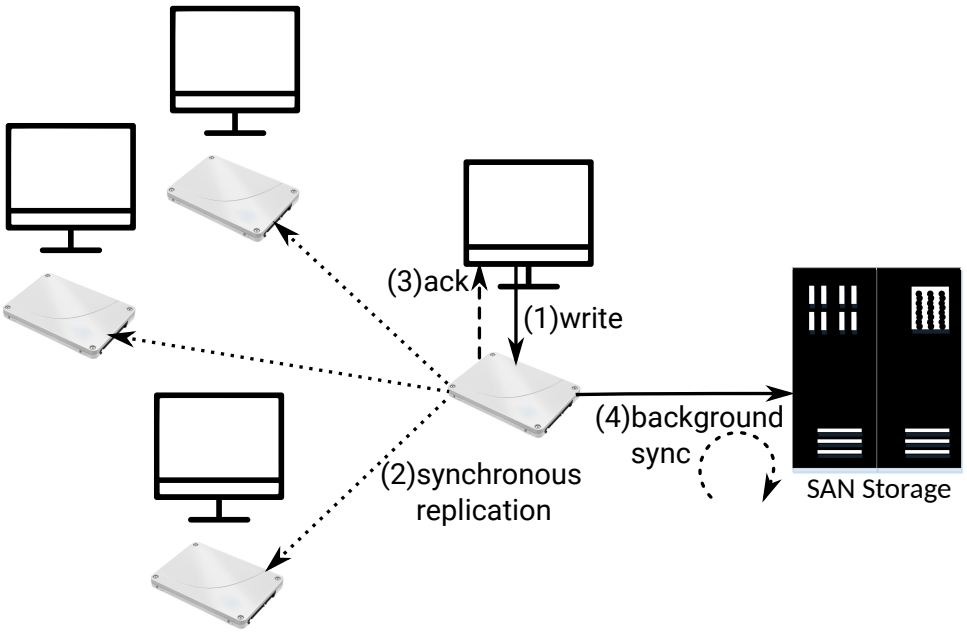

Figure 8: Fault-tolerant Write Back Policy from [4]

Advantages The major advantage of this policy is that synchronous mirroring of writes on peer host caches adds fault tolerance to the written data. The setup can survive `k` flash failures and `k+1` host failures. Even if one host is up, dirty block data can be written to the networked storage. If all peer hosts go down, even then the flash device can be extracted and attached to a 'surrogate' host that can flush the dirty data.

Disadvantages Write back with peering has a couple of possible disadvantages that the paper has not addressed. The biggest one is the space overhead of having writes replicated on flash caches across hosts. Another concern is the time needed to do synchronous replication to all the peers and how it compares to the time for writing to storage directly. The peers themselves might be located at different distances from the host in the data center.

3.3 Comparison of Various Policies

Figures 9, 10 and 11 show the comparison of the various policies with write back and write through. They are not comparabe directly because they come from different papers. Figure 9 shows the throughput achieved on various workloads normalized with respect to throughput obtained with write through. **FileBench (FB)** simulates a file server that hosting the home directories of a set of users; each user does a set of reads, writes, appends and deletes to the files in their respective home directories. **PostMark (PM)** simulates a file server running electronic mail. **TPC-C** is an OLTP workload that simulates an online retailer. **YCSB** is framework for comparing the performance of key-value stores. In this figure, it is used under three configurations — A (50:50 reads:writes), B (95:5 reads:writes) and F (50:50 reads:writes with read-write-modify behaviour). Journaled write back performs better than ordered write back due to write coalescing so has lesser overhead.

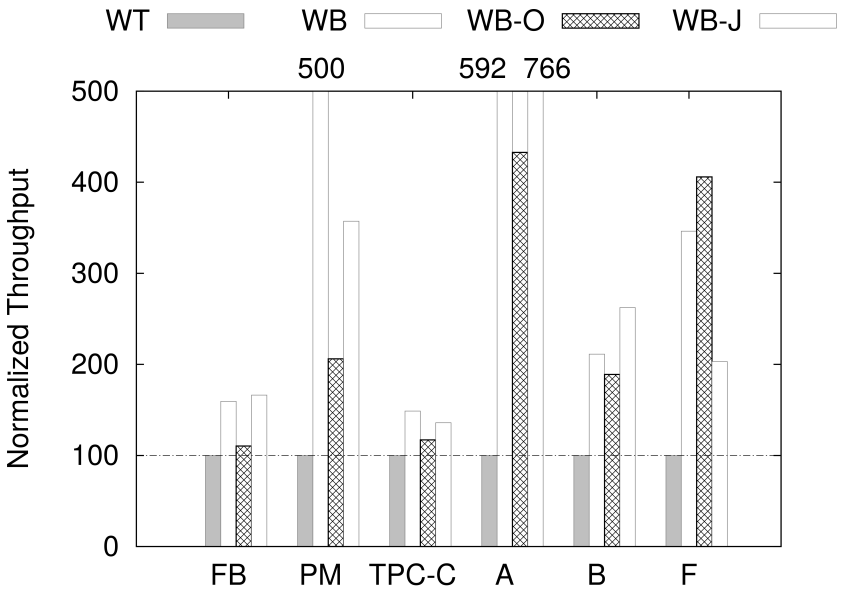

Figure 9: Performance Comparison of Ordered and Journaled Write Back with Write Through and Write Back (taken from [7])

Figure 10 shows the combined throughput of two VMs running two different workloads — Microsoft Exchange Jetstress and fio. **Jetstress** simulates a Microsoft Exchange Server database transactions with reads, writes, log writes etc. **fio** is a file I/O workload generator. As shown, the overhead of synchronously mirroring to two peer caches decreases the write throughput.

Figure 11 shows the comparison of write back flush and persist with write through and write back on three kinds of workloads — write-heavy (ms_nfs), read-heavy (webserver) and sync-heavy (varmail). Write back consistent is a variant of write back flush where writes are acknowledged immediately and the flushing happens in the background.

We now compare the various asynchronous write policies. We have not considered the scenario of hosts running as virtual machines. Virtual machine migrations can be tricky to handle with dirty writes still residing on the local flash of a physical machine while a VM is migrated to or started on another physical machine in case of a failure. Write back with peering [4] solves this problem by ensuring that the dirty data is fetched from the flash cache of some peer before resuming the VM.

Table 1 lists the comparison of the various caching policies on three axes — consistency achieved, cache space overhead and fault tolerance. The cache space overhead is just indicative and might not really be comparable as the policies come from different papers.

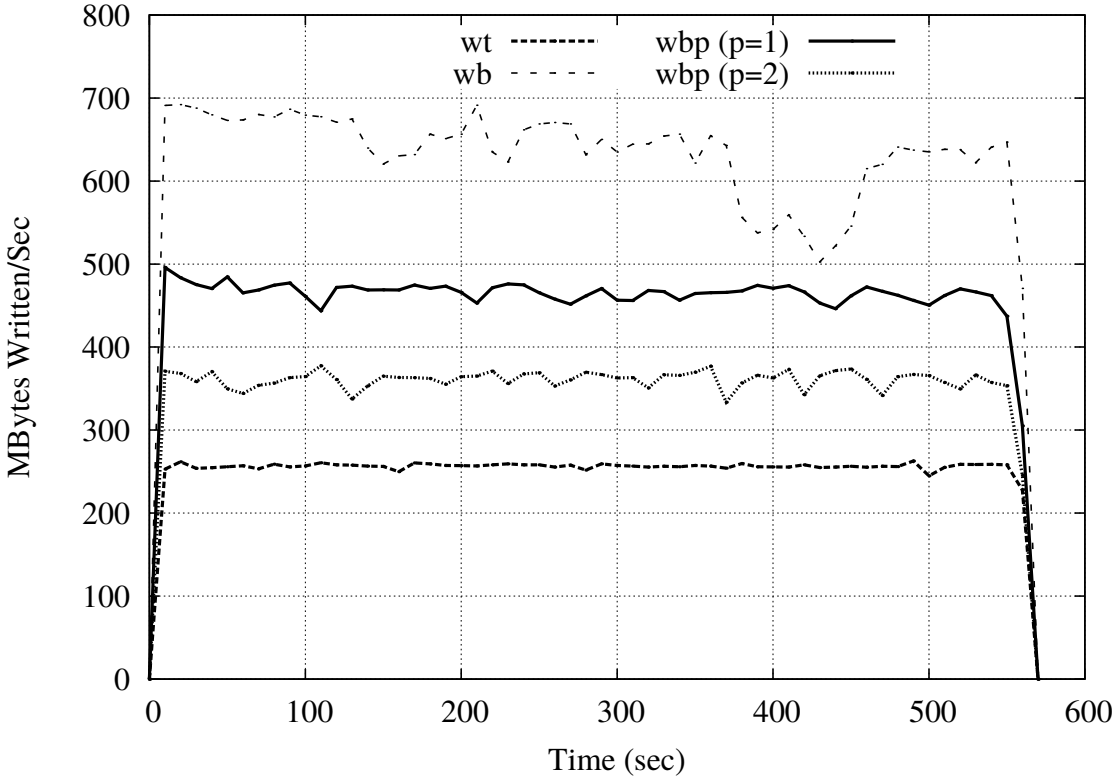

Figure 10: Performance Comparison of Write Back (with Peering) with Write Through and Write Back

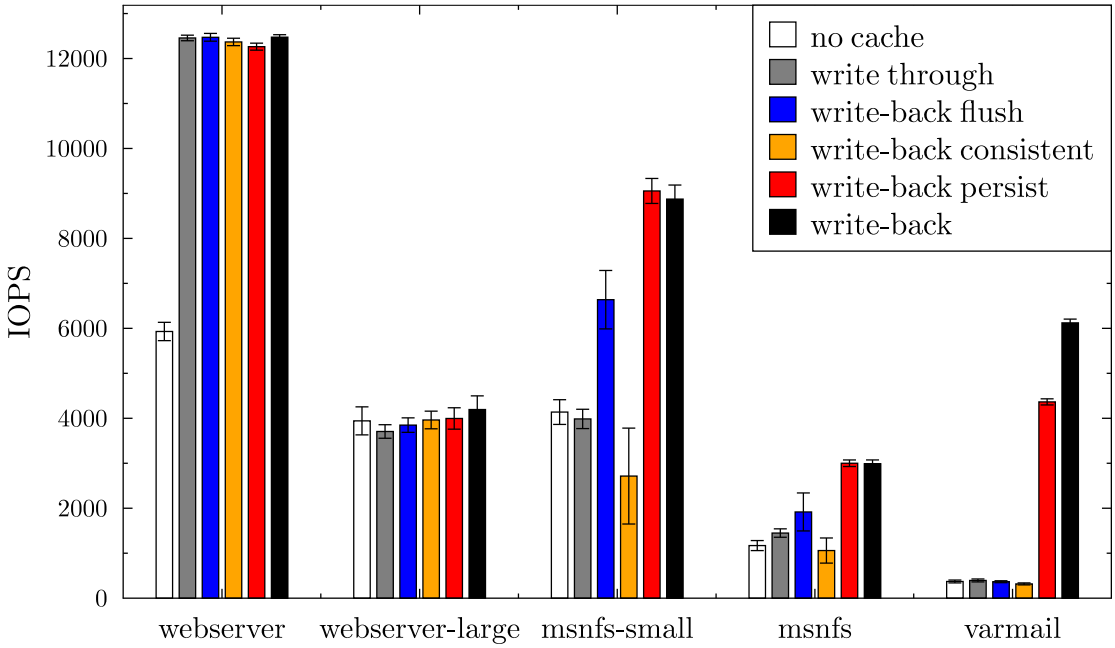

Figure 11: Performance Comparison of Write Back Flush and Write Back Persistent with Write Through and Write Back

4 WAN Remote Mirroring Solutions

4.1 Existing Approaches

Remote mirroring solutions across wide area networks (WAN) have parallels in the caching policies and the trade-offs involved therein.

Synchronous mirroring (like write through) trades off performance but ensures transactional consistency

Caching Policy	Consistency Provided	Cache Space Overhead	Fault Tolerance
Write through	Transactional		
Write back	Point-in-time	++	
Ordered write back	Point-in-time	++++	
Journaled write back	Point-in-time	+++	
Write back flush	Application-level	++	
Write back persist	Application-level	+++	Flash failures only
Write back with peering	Point-in-time	+++++++	Yes

Table 1: Comparison of various caching policies

i.e. no data is lost in case of disaster.

Semi-synchronous mirroring allows the application to continue as soon as data is written locally but sends it across in the background. The next write operation has to wait though if the acknowledgment of the previous remote write has not been received.

Asynchronous mirroring, on the other hand, not only gives immediate acknowledgment to the application, it also batches the data and sends it across periodically. The trade-off, then, is the amount of data loss that the organization can bear in case of a disaster.

4.2 The Middle Ground

Weatherspoon et al. [9] propose a novel approach that tries to find middle ground between semi-synchronous and synchronous remote replication. Their solution is built around the idea of using packet-level forward error correction (FEC) over WAN links. The key insight in their work is that there are non-congestion losses in the long haul WAN links. Thus they use packet-level FEC to effectively overcome packet losses and avoid triggering TCP congestion control mechanisms which can be detrimental to the rate of mirroring. Their mirroring solution is based on previous work described below.

4.2.1 FEC on edge routers

Balakrishnan et al. [3] propose a network device (called Maelstrom) that sits at the edge of the network and generates redundant packets using FEC for each outgoing flow on the WAN link, which can be used by a similar device at the other end to recover lost packets. This device also has a callback mechanism by which the egress router can notify the sending node that the redundant packets have been put on the network.

Figure 12 shows the principal components of the setup. The sending/receiving hosts are part of different networks connected by WAN link. The edge appliance in the sender's network (the egress router) is responsible for generating redundant packets using FEC and the one in the receiver's network (the ingress router) is responsible for recovery, if any. The redundant FEC packets are sent over a UDP channel to the edge appliance on the other end of the WAN link.

Advantages of Maelstrom's FEC approach

1. FEC ensures recovery of lost packets at the receiving end. Thus the re-transmission time is saved over the long haul link.

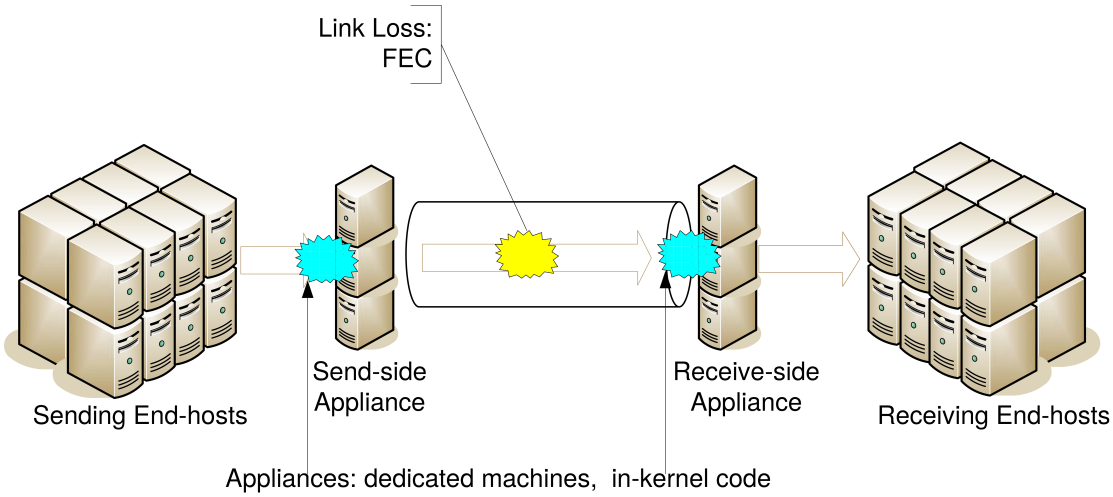

Figure 12: Maelstrom (adapted from [3])

2. Maelstrom's choice of doing the FEC encoding on network appliance for all the flows instead of doing it on the end hosts is a key contribution of their work. The writes are acknowledged to the application only after the egress router does a callback to notify that the packets have been put on the channel. This ensures that the writes are delivered across with a high probability even after a disaster at the primary site. Thus the only scope for loss is the packets which haven't left the egress router yet. But those writes are not acknowledged to the application yet, so the application's view is still consistent.

5 The Case for Probabilistic Asynchronous Writes

5.1 Motivation

The motivation for exploring a probabilistic write mechanism stems from the need to go as close to asynchronous write replication as possible in terms of performance, while still giving consistency and fault tolerance guarantees of a synchronous solution. We believe there is space for another asynchronous write policy which does better than the policies write back ordered and journaled [7], write back (with peering) [4] and write back flush and persist [8] discussed in section 3.

Ordered write back and *Journaled write back* [7] provide point-in-time consistency at best. The amount of data lost in case of a failure depends upon the rate of eviction of writes from the cache, which, in the best case can approach write through but might swamp the network. Ordered write back also needs a large cache space because of the need for evicting each write in order which in turn requires a new cache block for each update (even to the same block).

Write back flush and *Write back persist* [8] guarantee application-level consistency and the claim is that applications do not expect any guarantees except at write barriers. But this is true for specialized applications like databases which have mechanisms to recover from failures built into the application itself. Data can still be lost even if the storage is application-level consistent.

Write back with peering guarantees that writes do not get lost. But it comes at a high price of synchronous replication to the flash caches of k other peers. This degree of replication then decides the probability of the write surviving a single or multiple failures.

The idea that we are exploring is to have a way in which a write is immediately acknowledged after being done on flash while it is sent across to the networked storage asynchronously and it should reach there with a high probability so that data is not lost in case of a failure. The gist of our approach is to be able to estimate the probability of failure of packet delivery and trying to overcome it. This has multiple advantages — we are trying to get as close to synchronous write semantics as possible, while still being able to leverage the advantages associated with asynchronous writes.

5.2 System Components

We look at the various components of the system to find out what could go wrong and try to formulate a problem statement that encompasses possible solutions.

(a) Host failure.

(b) Flash cache failure.

(c) Networked storage failure.

(d) Network failure.

We focus on network failure probabilities and look at ways to overcome them for asynchronous writes to succeed with a high probability in case of failure. If the host or flash layer fails, the applications come to halt but as long as the storage is in a consistent state, the application can be restarted after restoring the host, or, if the application is running on a VM, starting the VM on a different host. We do not address the case of network storage failures.

To analyze the network failure probabilities, we look at three categories of papers — analytical modeling papers [2], empirical analysis papers [5, 6] and papers on routing in data center networks [1].

5.2.1 Data Center Networks

Typical data center networks are organized as fat trees. A fat tree is a multi-rooted tree organization which uses multiple interconnects between commodity switches. There are three layers of switches — core switches, aggregation switches and top-of-rack (TOR) switches. A fat tree built using k-port commodity switches will have *k pods*. Each pod has *k/2 TOR switches* and *k/2 aggregation switches*. Each TOR switch is connected to *k/2 end hosts* and *all the k/2 aggregation switches in its pod*. There are $(k/2)^2$ core switches. Each core switch is connected to each of the k pods. Figure 13 shows a typical fat tree network.

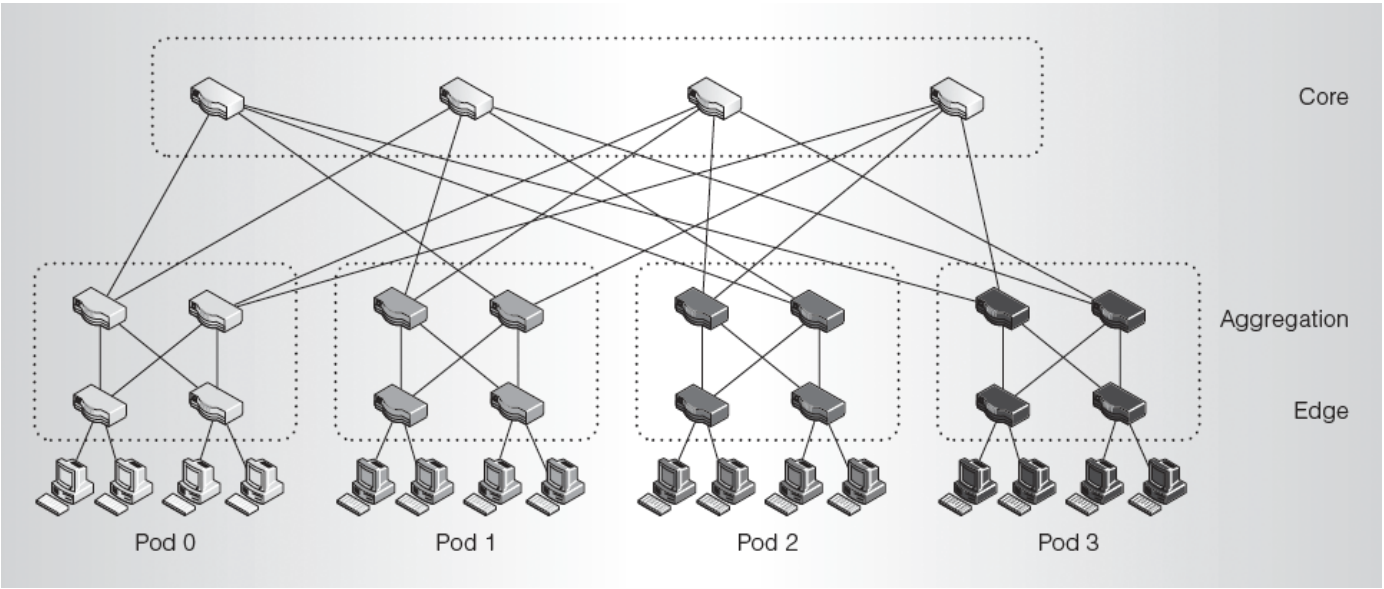

Figure 13: A fat tree network made of 4-port switches (adapted from [1])

Fat trees have multiple paths connecting any two nodes. Data center networks take advantage of this to distribute flows across routes. Equal cost multipath (ECMP) is one such protocol that routes flows statically to different paths based on hashes of certain fields in the IP header. Al-Fares et al. [1] propose a solution for the end-to-end routing of a network flow by proactively monitoring the network and allocating

alternate paths to flows that might be limited by the network bandwidth. Their key contribution lies in the out-of-band monitoring of the network to get a global view and taking decisions to change the routes taken by flows between end hosts.

Alshahrani et al. [2] propose an analytical model for the data center network to determine the delay and throughput of data center networks. They model the switches in the data center as a Jackson Queuing network focusing only on upstream traffic — going from TOR switches to core switches.

Gill et al. [5] look at the empirical data from data center networks and try to analyze the reliability of the various components of the data center networks e.g. links, aggregation switches, TOR switches, load balancers etc. Kandula et al. [6] do a similar analysis on the data center traffic.

5.3 Possible Solution Approaches

We propose to explore three variants of the solution approach. Although the underlying theme remains the same — acknowledging the writes immediately and then sending the updates across asynchronously with some probabilistic guarantees. Figure 14 shows the overall solution approach with step (3) being one of the three variants. The multiple outgoing arrows in the figure denotes the sending of redundant packets across multiple paths to ensure delivery to the end host, as suggested in variants 1 and 3 below.

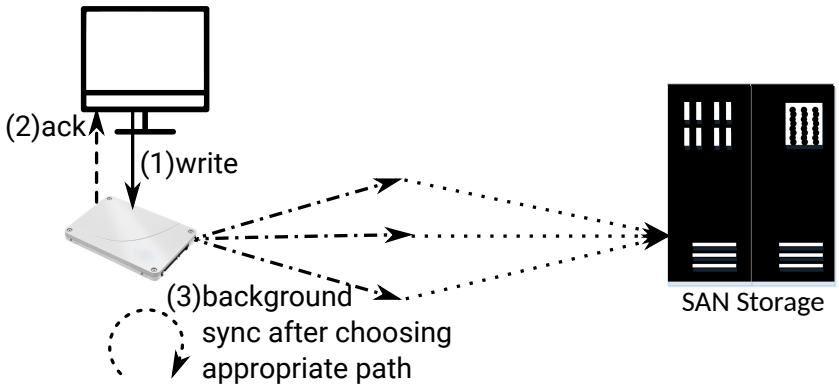

Figure 14: Our solution approach

5.3.1 Variant 1 – Global View of the Network

We propose to construct a global view of the data center network which comprises of the congestion on each path and loss characteristics. This could be done using a combination of out-of-band monitoring like [1] and analytical modeling like [2]. This global view then helps take decisions like sending a packet on a single path or sending multiple redundant packets on different paths to ensure its arrival.

5.3.2 Variant 2 – Use of Forward Error Correction

We propose to evaluate the use of FEC for correcting errors and ensuring reliable packet delivery. Yu et al. [12] have done some preliminary analysis of how FEC could be effective in overcoming packet losses.

5.3.3 Variant 3 – Use of Gossip Protocols

Although we have not covered gossip protocols, we propose to study and evaluate their usefulness in our scenario where time bounded gossip could be the answer to reliable packet delivery.

6 Conclusion

In this seminar, we looked at variants of asynchronous write back caching policies for host-side flash caches. We explored the pros and cons of each one of them and proposed a solution providing probabilistic guarantees on the success of asynchronous writes. We looked at papers from WAN replication and mirroring literature and data center networking to study components of our own problem and get ideas for possible solution approaches. Our reading led us to believe that there is space for implementing a new asynchronous write back caching policy which could be pushed even closer to synchronous writes without taking a performance hit. We plan to follow up on these proposed solutions in the Master's Thesis Project and present the results of our findings.

6.1 Acknowledgments

The idea of exploring this topic was proposed by Prof. Raju Rangaswami of Florida International University, who was a visiting faculty member at IIT Bombay in Autumn 2015. He was helpful in his feedback and comments through various online video conferences we had with him. Prof. Purushottam Kulkarni, my guide, was always available to help and I learned a lot under his guidance. We had long discussions during the seminar meetings which proved to be very helpful in shaping the final story for this seminar.

References

[1] Al-Fares, M., Radhakrishnan, S., Raghavan, B., Huang, N., and Vahdat, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2010), NSDI'10, USENIX Association, p. 19.

[2] Alshahrani, R., and Peyravi, H. Modeling and Simulation of Data Center Networks. In *Proceedings of the 2Nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation* (New York, NY, USA, 2014), SIGSIM PADS '14, ACM, pp. 75–82.

[3] Balakrishnan, M., Marian, T., Birman, K., Weatherspoon, H., and Vollset, E. Maelstrom: Transparent Error Correction for Lambda Networks. *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI 08)* (2008), 263–277.

[4] Bhagwat, D., Patil, M., Ostrowski, M., Vilayannur, M., Jung, W., and Kumar, C. A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 287–300.

[5] Gill, P., Jain, N., and Nagappan, N. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. *SIGCOMM Computer Communication Review 41*, 4 (aug 2011), 350–361.

[6] Kandula, S., Sengupta, S., Greenberg, A., Patel, P., and Chaiken, R. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference* (New York, NY, USA, 2009), IMC '09, ACM, pp. 202–208.

[7] Koller, R., Marmol, L., Rangaswami, R., Sundararaman, S., Talagala, N., and Zhao, M. Write Policies for Host-side Flash Caches. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, 2013), USENIX, pp. 45–58.

[8] Qin, D., Brown, A. D., and Goel, A. Reliable Writeback for Client-side Flash Caches. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14)* (Philadelphia, PA, 2014), USENIX Association, pp. 451–462.

[9] Weatherspoon, H., Ganesh, L., Marian, T., Balakrishnan, M., and Birman, K. Smoke and Mirrors: Reflecting Files at a Geographically Remote Location Without Loss of Performance. *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST 09)* (2009), 211–224.

[10] Memory hierarchy - wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory_hierarchy. Last accessed April 14, 2016.

[11] On hyper-convergence. https://blog.cdemi.io/on-hyper-convergence/. Last accessed April 15, 2016.

[12] Yu, X., Modestino, J. W., Kurceren, R., and Chan, Y. S. A Model-based Approach to Evaluation of the Efficacy of FEC Coding in Combating Network Packet Losses. *IEEE/ACM Trans. Netw. 16*, 3 (jun 2008), 628–641.