Analysis of HDFS under HBase

A Facebook Messages Case Study

TYLER HARTER, DHRUBA BORTHAKUR, SIYING DONG, AMITANAND AIYER, LIYIN TANG, ANDREA C. ARPACI-DUSSEAU, AND REMZI H. ARPACI-DUSSEAU

Tyler Harter is a student at the University of Wisconsin—Madison, where he has received his bachelor's and master's degrees and is currently pursuing a computer science Ph.D. Professors Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau advise him, and he is funded by an NSF Fellowship. Tyler is interested in helping academics understand the I/O workloads seen in industry; toward this end he has interned at Facebook twice and studied the Messages application as a Facebook Fellow. Tyler has also studied desktop I/O, co-authoring an SOSP '11 paper, "A File Is Not a File...," which received a Best Paper award. harter@cs.wisc.edu

Dhruba Borthakur is an engineer in the Database Engineering Team at Facebook. He is leading the design of RocksDB, a datastore optimized for storing data in fast storage. He is one of the founding architects of the Hadoop Distributed File System and has been instrumental in scaling Facebook's Hadoop cluster to multiples of petabytes. Dhruba also is a contributor to the Apache HBase project and Andrew File System (AFS). Dhruba has an MS in computer science from the University of Wisconsin-Madison. He hosts a Hadoop blog at http://hadoopblog.blogspot.com/ and a RocksDB blog at http://rocksdb.blogspot.com. dhruba@fb.com

Large-scale distributed storage systems are exceedingly complex and time-consuming to design, implement, and operate. As a result, rather than cutting new systems from whole cloth, engineers often opt for layered architectures, building new systems upon already-existing ones to ease the burden of development and deployment. In this article, we examine how layering causes write amplification when HBase is run on top of HDFS and how tighter integration could result in improved write performance. Finally, we take a look at whether it makes sense to include an SSD to improve performance while keeping costs in check.

Layering, as is well known, has many advantages [6]. For example, construction of the Frangipani distributed file system was greatly simplified by implementing it atop Petal, a distributed and replicated block-level storage system [7]. Because Petal provides scalable, fault-tolerant virtual disks, Frangipani could focus solely on file-system-level issues (e.g., locking); the result of this two-layer structure, according to the authors, was that Frangipani was "relatively easy to build."

Unfortunately, layering can also lead to problems, usually in the form of decreased performance, lowered reliability, or other related issues. For example, Denehy et al. show how naïve layering of journaling file systems atop software RAIDs can lead to data loss or corruption [2]. Similarly, others have argued about the general inefficiency of the file system atop block devices [4].

In this article, we focus on one specific, and increasingly common, layered storage architecture: a distributed database (HBase, derived from Google's BigTable) atop a distributed file system (HDFS, derived from the Google File System). Our goal is to study the interaction of these important systems with a particular focus on the lower layer, which leads to our highest-level question: Is HDFS an effective storage back end for HBase?

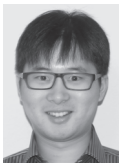
To derive insight into this hierarchical system, and therefore answer this question, we trace and analyze it under a popular workload: Facebook Messages (FM). FM is a messaging system that enables Facebook users to send chat and email-like messages to one another; it is quite popular, handling millions of messages each day. FM stores its information within HBase (and thus, HDFS) and hence serves as an excellent case study.

To perform our analysis, we collected detailed HDFS traces over an eight-day period on a subset of FM machines. These traces reveal a workload very unlike traditional GFS/HDFS patterns. Whereas workloads have traditionally consisted of large, sequential I/O to very large files, we find that the FM workload represents the opposite. Files are small (750 KB median), and I/O is highly random (50% of read runs are shorter than 130 KB).

We also use our traces to drive a multilayer simulator, allowing us to analyze I/O patterns across multiple layers beneath HDFS. From this analysis, we derive numerous insights. For

Siying Dong is a software engineer working in the Database Engineering team at Facebook, focusing on RocksDB. He also worked on Hive, HDFS, and some other data warehouse infrastructures. Before joining Facebook, Siying worked in the SQL Azure Team at Microsoft. He received a bachelor's degree from Tsinghua University and a master's degree from Brandeis University. siying.d@fb.com

Amitanand Aiyer is a research scientist in the Core Data team at Facebook. He focuses on building fault-tolerant distributed systems and has been working on HBase to improve its availability and fault tolerance. Amitanand is an HBase Committer and has a Ph.D. from the University of Texas at Austin. amitanand.s@fb.com



Liyin Tang is a software engineer from the Core Data team at Facebook, where he focuses on building highly available and reliable storage services, and helps the service scale in the face of exponential data growth. Liyin also works as an HBase PMC member in the Apache community. He has a master's degree in computer science from the University of Southern California and a bachelor's degree in software engineering from Shanghai Jiao Tong University. liyin.tang@fb.com

example, we find that many features at different layers amplify writes, and that these features often combine multiplicatively. For example, HBase logs introduced a 10x overhead on writes, whereas HDFS replication introduced a 3x overhead; together, these features produced a 30x write overhead. When other features such as compaction and caching are also considered, we find writes are further amplified across layers. At the highest level, writes account for a mere 1% of the baseline HDFS I/O, but by the time the I/O reaches disk, writes account for 64% of the workload.

This finding indicates that even though FM is an especially read-heavy workload within Facebook, it is important to optimize for both reads and writes. We evaluate potential optimizations by modeling various hardware and software changes with our simulator. For reads, we observe that requests are highly random; therefore, we evaluate using flash to cache popular data. We find that adding a small SSD (e.g., 60 GB) can reduce latency by 3.5x. For writes, we observe compaction and logging are the major causes (61% and 36%, respectively); therefore, we evaluate HDFS changes that give HBase special support for these operations. We find such HDFS specialization yields a 2.7x reduction in replication-related network I/O and a 6x speedup for log writes. More results and analysis are discussed in our FAST '14 paper [8].

Background and Methodology

The FM storage stack is based on three layers: distributed database over distributed file system over local storage. These three layers are illustrated in Figure 1 under "Actual Stack." As shown, FM uses HBase for the distributed database and HDFS for the distributed file system. HBase provides a simple API allowing FM to put and get key-value pairs. HBase stores these records in data files in HDFS. HDFS replicates the data across three machines and thus can handle disk and machine failures. By handling these low-level fault tolerance details, HDFS frees HBase to focus on higher-level database logic. HDFS in turn stores replicas of HDFS blocks as files in local file systems. This design enables HDFS to focus on replication while leaving details such as disk layout to local file systems. The primary advantage of this layered design is that each layer has only a few responsibilities, so each layer is simpler (and less bug prone) than a hypothetical single system that would be responsible for everything.

One important question about this layered design, however, is: What is the cost of simplicity (if any) in terms of performance? We explore this question in the context of the FM workload. To understand how FM uses the HBase/HDFS stack, we trace requests from HBase to HDFS, as shown in the Figure 1. We collect traces by deploying a new HDFS tracing framework that we built to nine FM machines for 8.3 days, recording 71 TB of HDFS I/O.

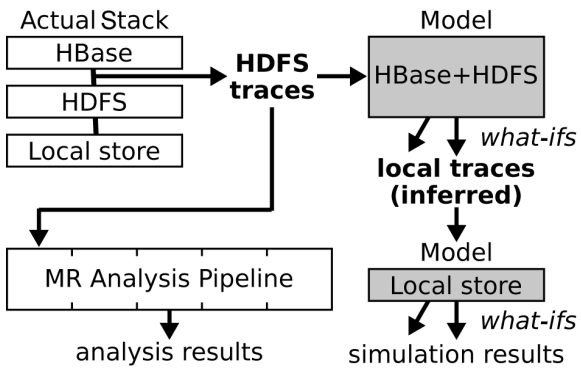


Figure 1: Tracing, analysis, and simulation

The traces record the I/O of four HBase activities that use HDFS: logging, flushing, reading, and compacting. When HBase receives a put request, it immediately logs the record to an HDFS file for persistence. The record is also added to an HBase write buffer, which, once filled, HBase flushes to a sorted data file. Data files are never modified, so when a get request arrives, HBase reads multiple data files in order to find the latest version of the data. To limit the number of files that must be read, HBase occasionally compacts old files, which involves merge sorting multiple small data files into one large file and then deleting the small files.

We do two things with our traces of these activities. First, as Figure 1 shows, we feed them to a pipeline of MapReduce analysis jobs. These jobs compute statistics that characterize the workload. We discuss

Analysis of HDFS under HBase: A Facebook Messages Case Study

Andrea Arpaci-Dusseau is a professor and associate chair of computer sciences at the University of Wisconsin—Madison. Andrea co-leads a research group with her husband, Remzi, and has advised 14 students through their Ph.D. dissertations; their group has received many Best Paper awards and sorting world records. She is a UW—Madison Vilas Associate and received the Carolyn Rosner "Excellent Educator" award; she has served on the NSF CISE Advisory Committee and as faculty co-director of the Women in Science and Engineering (WISE) Residential Learning Community. dusseau@cs.wisc.edu

Remzi Arpaci-Dusseau is a full professor in the Computer Sciences Department at the University of Wisconsin—Madison. Remzi co-leads a research group with his wife, Andrea; details of their research can be found at http://research.cs.wisc.edu/adsl. Remzi has won the SACM Professor-of-the-Year award four times and the Rosner "Excellent Educator" award once. Chapters from a freely available OS book he and his wife co-wrote, found at www.ostep.org, have been downloaded over 1/2 million times. Remzi has served as co-chair of USENIX ATC, FAST, OSDI, and (upcoming) SOCC conferences. Remzi has been a NetApp faculty fellow, an IBM faculty award winner, an NSF CAREER award winner, and currently serves on the Samsung DS CTO and UW Office of Industrial Partnership advisory boards. remzi@cs.wisc.edu

these characteristics in the next section and suggest ways to improve both the hardware and software layers of the stack. Second, we evaluate our suggestions via a simulation of layered storage. We feed our traces to a model of HBase and HDFS that translates the HDFS traces into inferred traces of requests to local file systems. For example, our simulator translates an HDFS write to three local file-system writes based on a model of triple replication. We then feed our inferred traces of local file-system I/O to a model of local storage. This model computes request latencies and other statistics based on submodels of RAM, SSDs, and rotational disks (each with its own block scheduler). We use these models to evaluate different ways to build the software and hardware layers of the stack.

Workload Behavior

In this section, we characterize the FM workload with four questions: What activities cause I/O at each layer of the stack? How large is the dataset? How large are HDFS files? And, is I/O sequential?

I/O Activities

We begin by considering the number of reads and writes at each layer of the stack in Figure 2. The first bar shows HDFS reads and writes, excluding logging and compaction overheads. At this level, writes represent only 1% of the 47 TB of I/O. The second bar includes these overheads. As shown, overheads are significant and write dominated, bringing the writes to 21%.

HBase tolerates failures by replicating data with HDFS. Thus, one HDFS write causes three writes to local files and two network transfers. The third bar of Figure 2 shows that this tripling increases the writes to 45%. Not all this file-system I/O will hit disk, as OS caching absorbs some of the reads. The fourth bar shows that only 35 TB of disk reads are caused by the 56 TB of file-system reads. The bar also shows a write increase, as very small file-system writes cause 4 KB-block disk writes. Because of these factors, writes represent 64% of disk I/O.

Dataset Size

Figure 3 gives a layered overview similar to that of Figure 2, but for data rather than I/O. The first bar shows 3.9 TB of HDFS data received some non-overhead I/O during tracing (data deleted during tracing is not counted). Nearly all this data was read and a small portion written. The second bar shows data touched by any I/O (including compaction and logging overheads). The third bar shows how much data is touched at the local level during tracing. This bar also shows untouched data. Most of the 120 TB of data is very cold; only a third is accessed over the eight-day period.

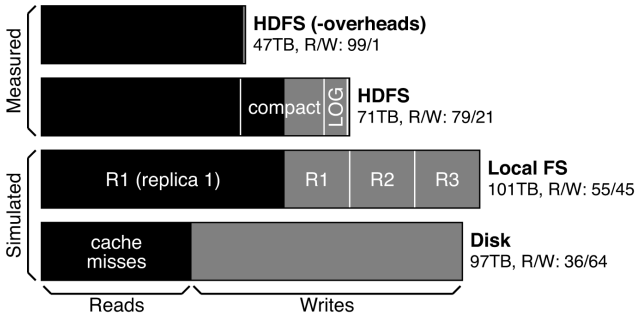


Figure 2: I/O across layers. Black sections represent reads and gray sections represent writes. The top two bars indicate HDFS I/O as measured directly in the traces. The bottom two bars indicate local I/O at the file-system and disk layers as inferred via simulation.

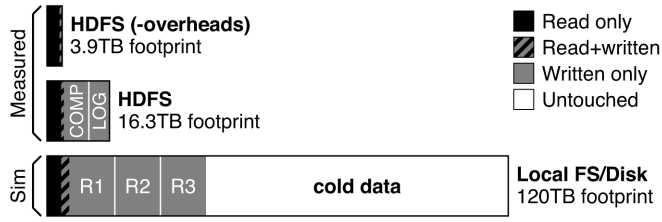


Figure 3: Data across layers. This is the same as Figure 2 but for data instead of I/O. COMP is compaction.

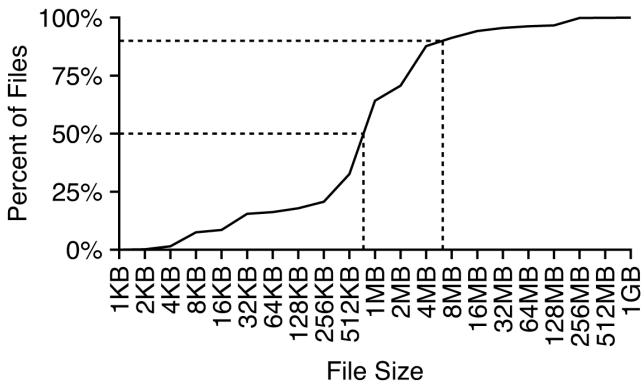


Figure 4: File-size CDF. The distribution is over the sizes of created files with 50th and 90th percentiles marked.

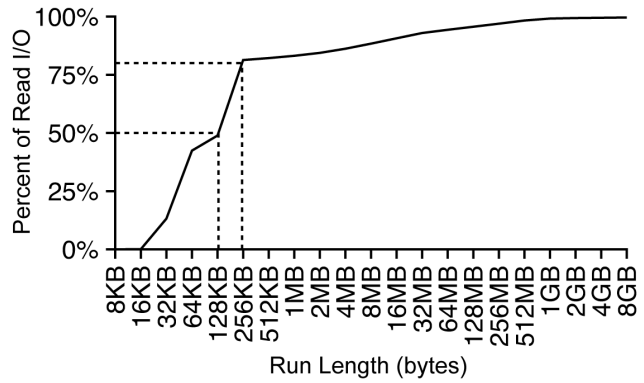


Figure 5: Run-size CDF. The distribution is over sequential read runs, with 50th and 80th percentiles marked.

File Size

GFS (the inspiration for HDFS) assumed "multi-GB files are the common case, and should be handled efficiently" [5]. Previous HDFS workload studies also show this; for example, MapReduce inputs were found to be about 23 GB at the 90th percentile (Facebook in 2010) [1].

Figure 4 reports a CDF of the sizes of HDFS files created during tracing. We observe that created files tend to be small; the median file is 750 KB, and 90% are smaller than 6.3 MB. This means that the data-to-metadata ratio will be higher for FM than for traditional workloads, suggesting that it may make sense to distribute metadata instead of handling it all with a single NameNode.

Sequentiality

GFS is primarily built for sequential I/O and, therefore, assumes "high sustained bandwidth is more important than low latency" [5]. All HDFS writes are sequential, because appends are the only type of writes supported, so we now measure read sequentiality. Data is read with sequential runs of one or more contiguous read requests. Highly sequential patterns consist of large runs, whereas random patterns consist mostly of small runs.

Figure 5 shows a distribution of read I/O, distributed by run size. We observe that most runs are fairly small. The median run size is 130 KB, and 80% of runs are smaller than 250 KB, indicating FM reads are very random. These random reads are primarily caused by get requests; the small (but significant) portion of reads that are sequential are mostly due to compaction reads.

Layering: Pitfalls and Solutions

In this section, we discuss different ways to layer storage systems and evaluate two techniques for better integrating layers.

Layering Background

Three important layers are the local layer (e.g., disks, local file systems, and a DataNode), the replication layer (e.g., HDFS), and the database layer (e.g., HBase). FM composes these in a mid-replicated pattern (Figure 6a), with the database above replication and the local stores below. The merit of this design is simplicity. The database can be built with the assumption that underlying storage will be available and never lose data. Unfortunately, this approach separates computation from data. Computation (e.g., compaction) can co-reside with, at most, one replica, so all writes involve network I/O.

Top-replication (Figure 6b) is an alternative used by Salus [9]. Salus supports the HBase API but provides additional robustness and performance advantages. Salus protects against memory corruption by replicating database computation as well as the data itself. Doing replication above the database level also reduces network I/O. If the database wants to reorganize data on disk (e.g., via compaction), each database replica can do so on its local copy. Unfortunately, top-replicated storage is complex, because the database layer must handle underlying failures as well as cooperate with other databases.

Mid-bypass (Figure 6c) is a third option proposed by Zaharia et al. [10]. This approach (like mid-replication) places the replication layer between the database and the local store; but, to improve performance, an RDD (Resilient Distributed Dataset)

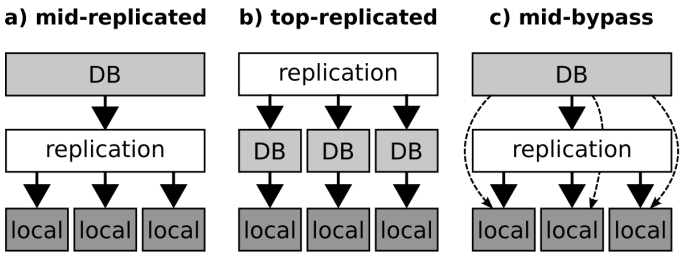


Figure 6: Layered architectures. The HBase architecture (mid-replicated) is shown, as well as two alternatives. Top-replication reduces network I/O by co-locating database computation with database data. The mid-bypass architecture is similar to mid-replication but provides a mechanism for bypassing the replication layer for efficiency.

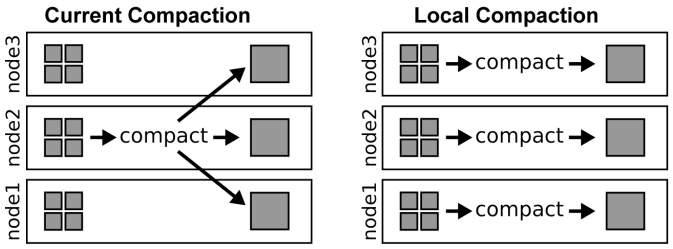


Figure 7: Local-compaction architecture. The HBase architecture (left) shows how compaction currently creates a data flow with significant network I/O, represented by the two lines crossing machine boundaries. An alternative (right) shows how local reads could replace network I/O.

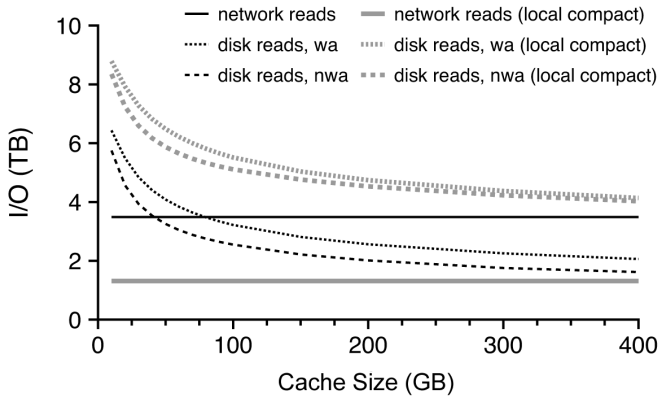


Figure 8: Local-compaction results. The thick gray lines represent HBase with local compaction, and the thin black lines represent HBase currently. The solid lines represent network reads, and the dashed lines represent disk reads; long-dash represents the no-write allocate cache policy and short-dash represents write allocate.

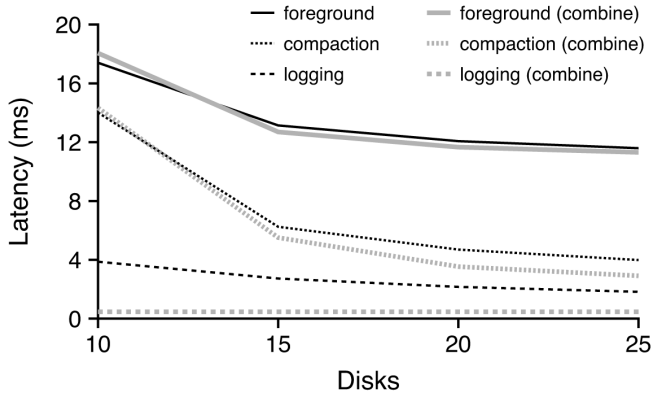


Figure 9: Combined logging results. Disk latencies for various activities are shown, with (gray) and without (black) combined logging.

API lets the database bypass the replication layer. Network I/O is avoided by shipping computation directly to the data. HBase compaction, if built upon two RDD transformations, join and sort, could avoid much network I/O.

Local Compaction

We simulate the mid-bypass approach, shipping compaction operations directly to all the replicas of compaction inputs. Figure 7 shows how local compaction differs from traditional compaction; network I/O is traded for local I/O, to be served by local caches or disks.

Figure 8 shows the result: a 62% reduction in network reads, from 3.5 TB to 1.3 TB. The figure also shows disk reads, with and without local compaction, and with either write allocate (wa) or no-write allocate (nwa) caching policies. We observe that disk I/O increases slightly more than network I/O decreases. For example, with a 100-GB cache, network I/O is decreased by 2.2 GB, but disk reads are increased by 2.6 GB for no-write allocate. This is unsurprising: HBase uses secondary replicas for fault tolerance rather than for reads, so secondary replicas are written once (by a flush or compaction) and read at most once (by compaction). Thus, local-compaction reads tend to (1) be misses and (2) pollute the cache with data that will not be read again. Even still, trading network I/O for disk I/O in this way is desirable, as network infrastructure is generally much more expensive than disks.

Combined Logging

We now consider the interaction between replication and HBase logging. Currently, a typical HDFS DataNode receives logs from three RegionServers. Because HDFS just views these logs as regular HDFS files, HDFS will generally write them to different disks. We evaluate an alternative to this design: combined logging. With this approach, HBase passes a hint to HDFS, identifying the logs as files that are unlikely to be read back. Given this hint, HDFS can choose a write-optimized layout for the logs. In particular, HDFS can interleave the multiple logs in a single write-stream on a single dedicated disk.

We simulate combined logging and measure performance for requests that go to disk; we consider latencies for logging, compaction, and foreground reads. Figure 9 reports the results for varying numbers of disks. The latency of log writes decreases dramatically with combined logging (e.g., by 6x with 15 disks). Foreground-read and compaction requests are also slightly faster in most cases due to less competition with logs for seeks. Currently, put requests do not block on log writes, so logging is a background activity. If, however, HBase were to give a stronger guarantee for puts (namely that data is durable before returning), combined logging would make that guarantee much cheaper.

Hardware: Adding a Flash Layer

Earlier, we saw FM has a very large, mostly cold dataset; keeping all this data in flash would be wasteful, costing upwards of $10k/machine (assuming flash costs $0.80/GB). Thus, because we should not just use flash, we now ask, should we use no flash or some flash? To answer this, we need to compare the performance improvements provided by flash to the corresponding monetary costs.

We first estimate the cost of various hardware combinations (assuming disks are $100 each, RAM costs $5/GB, and flash costs $0.8/GB). To compute performance, we run our simulator on our traces using each hardware combination. We try nine RAM/disk combinations, each with no flash or a 60 GB SSD; these represent three amounts of disk and three amounts of RAM. When the 60 GB SSD is used, the RAM and flash function as a tiered LRU cache.

Figure 10 shows how adding flash changes both cost and performance. For example, the leftmost two bars of the leftmost plot show that adding a 60 GB SSD to a machine with 10 disks and 10 GB of RAM decreases latency by a factor of 3.5x (from 19.8 ms to 5.7 ms latency) while only increasing costs by 5%. Across all nine groups of bars, we observe that adding the SSD always increases costs by 2–5% while decreasing latencies by 17–71%. In two thirds of the cases, flash cuts latency by more than 40%.

We have shown that adding a small flash cache greatly improves performance for a marginal initial cost. Now, we consider long-term replacement caused by flash wear, as commercial SSDs often support only 10k program/erase cycles. We consider three factors that affect flash wear. First, if there is more RAM, there will be fewer evictions to the flash level of the cache and therefore fewer writes and less wear. Second, if the flash device is large, writes will be spread over more cells, so each cell will live longer. Third, using a strict LRU policy can cause excessive writes for some workloads by frequently promoting and evicting the same items back and forth between RAM and flash.

Figure 11 show how these three factors affect flash lifetime. The black "Strict LRU" lines correspond to the same configuration used in Figure 10 for the 10 GB and 30 GB RAM results. The amount of RAM makes a significant difference. For example, for strict LRU, the SSD will live 58% longer with 30 GB of RAM instead of 10 GB of RAM. The figure also shows results for a wear-friendly policy with the gray lines. In this case, RAM and flash are each an LRU independently, and RAM evictions are inserted into flash, but (unlike strict LRU) flash hits are not repromoted to RAM. This alternative to strict LRU greatly reduces wear by reducing movement between RAM and flash. For example, with 30 GB of RAM, we observe that 240 GB SSD will last 2.7x longer if the wear-friendly policy is used. Finally, the figure shows that the amount of flash is a major factor in

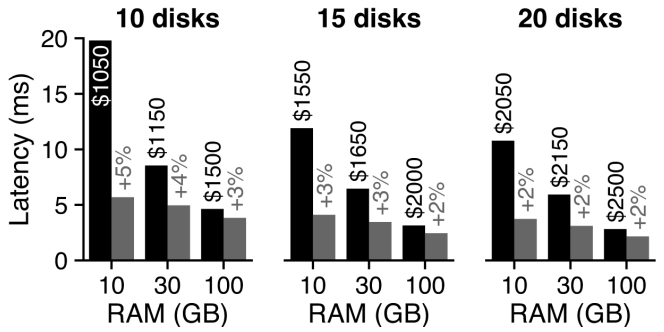


Figure 10: Flash cost and performance. Black bars indicate latency without flash, and gray bars indicate latency with a 60 GB SSD. Latencies are only counted for foreground I/O (e.g., servicing a get), not background activities (e.g., compaction). Bar labels indicate cost. For the black bars, the labels indicate absolute cost, and for the gray bars, the labels indicate the cost increase relative to the black bar.

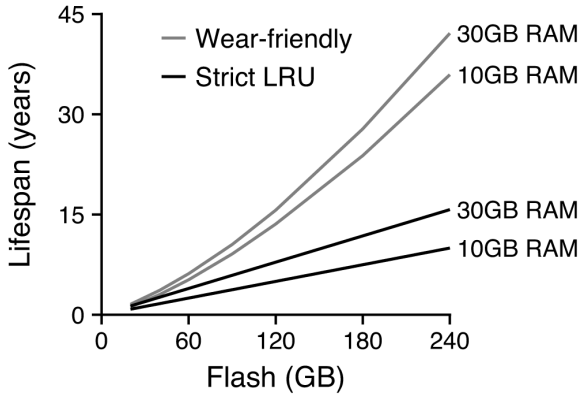


Figure 11: Flash lifetime. The relationship between flash size and flash lifetime is shown for both the keep policy (gray lines) and promote policy (black lines). There are two lines for each policy (10 or 30 GB RAM).

flash lifetime. Whereas the 20 GB SSD lasts between 0.8 and 1.6 years (depending on policy and amount of RAM), the 120 GB SSD always lasts at least five years.

We conclude that adding a small SSD cache is a cost-effective way to improve performance. Adding a 60 GB SSD can often double performance for only a 5% cost increase. We find that for large SSDs, flash has a significant lifetime, and so avoiding wear is probably unnecessary (e.g., 120 GB SSDs last more than five years with a wear-heavy policy), but for smaller SSDs, it is useful to choose caching policies that avoid frequent data shuffling.

Conclusions

We have presented a detailed multilayer study of storage I/O for Facebook Messages. Our research relates to common storage ideas in several ways. First, the GFS-style architecture is based on workload assumptions, such as "high sustained bandwidth is more important than low latency" and "multi-GB files are the

common case, and should be handled efficiently" [5]. We find FM represents the opposite workload, being dominated by small files and random I/O.

Second, layering storage systems is very popular; Dijkstra found layering "proved to be vital for the verification and logical soundness" of an OS [3]. We find, however, that simple layering has a cost. In particular, we show that relative to the simple layering currently used, tightly integrating layers reduces replication-related network I/O by 62% and makes log writes 6x faster. We further find that layers often amplify writes multiplicatively. For example, a 10x logging overhead (HBase level) combines with a 3x replication overhead (HDFS level), producing a 30x write overhead.

Third, flash is often extolled as a disk replacement. For example, Jim Gray has famously said that "tape is dead, disk is tape, flash is disk." For Messages, however, flash is a poor replacement for disk, as the dataset is very large and mostly cold, and storing it all in flash would cost over $10k/machine. Although we conclude that most data should continue to be stored on disk, we find small SSDs can be quite useful for storing a small, hot subset of the data. Adding a 60 GB SSD can often double performance while only increasing costs by 5%.

In this work, we take a unique view of Facebook Messages, not as a single system but as a complex composition of layered subsystems. We believe this perspective is key to deeply understanding modern storage systems. Such understanding, we hope, will help us better integrate layers, thereby maintaining simplicity while achieving new levels of performance.

Acknowledgments

We thank the anonymous reviewers and Andrew Warfield (our FAST shepherd) for their tremendous feedback, as well as members of our research group for their thoughts and comments on this work at various stages. We also thank Pritam Damania, Adela Maznikar, and Rishit Shroff for their help in collecting HDFS traces.

This material was supported by funding from NSF grants CNS-1319405 and CNS-1218405 as well as generous donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Tyler Harter is supported by the NSF Fellowship and Facebook Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other institutions.

References

[1] Yanpei Chen, Sara Alspaugh, and Randy Katz, "Interactive Analytical Processing in Big Data Systems: A Cross-Industry Study of MapReduce Workloads," *Proceedings of the VLDB Endowment*, vol. 5, no. 12 (August 2012), pp. 1802–1813.

[2] Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, "Journal-Guided Resynchronization for Software RAID," *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)* (December 2005), pp. 87–100.

[3] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *Communications of the ACM*, vol. 11, no. 5 (May 1968), pp. 341–346.

[4] Gregory R. Ganger, "Blurring the Line Between OSes and Storage Devices," Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)* (October 2003), pp. 29–43.

[6] Jerome H. Saltzer, David P. Reed, and David D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4 (November 1984), pp. 277–288.

[7] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee, "Frangipani: A Scalable Distributed File System," *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)* (October 1997), pp. 224–237.

[8] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, "Analysis of HDFS under HBase: A Facebook Messages Case Study," *Proceedings of the 12th Conference on File and Storage Technologies (FAST '14)* (February 2014).

[9] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin, "Robustness in the Salus Scalable Block Store," *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)* (April 2013).

[10] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-memory Cluster Computing," *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)* (April 2012).

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Publish and Present Your Work at USENIX Conferences

The program committees of the following conferences are seeking submissions. CiteSeer ranks the USENIX Conference Proceedings among the the top ten highest-impact publication venues for computer science.

Get more details about each of these Calls for Papers and Participation at www.usenix.org/cfp.

INFLOW '14: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads
October 5, 2014, Broomfield, CO
Submissions due: July 1, 2014, 11:59 p.m. PDT

The goal of INFLOW '14 is to bring together researchers and practitioners working in systems, across the hardware/software stack, who are interested in the cross-cutting issues of NVM/Flash technologies, operating systems, and emerging workloads.

HotDep '14: 10th Workshop on Hot Topics in System Dependability
October 5, 2014, Broomfield, CO
Submissions due: July 10, 2014

HotDep '14 will bring forth cutting-edge research ideas spanning the domains of systems and fault tolerance/reliability. The workshop will build links between the two communities and serve as a forum for sharing ideas and challenges.

FAST '15: 13th USENIX Conference on File and Storage Technologies
February 16–19, 2015, Santa Clara, CA
Submissions due: September 23, 2014, 9:00 p.m. PDT (Hard deadline, no extensions)

The 13th USENIX Conference on File and Storage Technologies (FAST '15) brings together storage-system researchers and practitioners to explore new directions in the design, implementation, evaluation, and deployment of storage systems. The program committee will interpret "storage systems" broadly; everything from low-level storage devices to information management is of interest. The conference will consist of technical presentations, including refereed papers, Work-in-Progress (WiP) reports, poster sessions, and tutorials.