Conference Reports

In this issue:

13th USENIX Conference on File and Storage Technologies

February 16–19, 2015, Santa Clara, CA

Summarized by Zhen Cao, Akhilesh Chaganti, Ming Chen, and Rik Farrow

Opening Remarks

Summarized by Rik Farrow

Erez Zadok opened the conference with the numbers: 130 submissions with 28 papers selected. They used a two round online review process, with each paper getting three reviews during the first round, and the 68 remaining papers getting two more reviews in the next round. The final decisions were made in an all-day meeting at Stony Brook.

Jiri Schindler announced awards, starting with ACM Test-of-Time Fast Track awards going to "RAIDShield: Characterizing, Monitoring, and Proactively Protecting against Disk Failures," by Ao Ma et al., and "BetrFS: A Right-Optimized Write-Optimized File System," by William Jannen et al. The Best Paper Award went to "Skylight—A Window on Shingled Disk Operation," by Abutalib Aghayev and Peter Desnoyers. These researchers cut a window into a shingled (SMR) drive so that they could use a high-speed camera to record disk seeks, an interesting form of reverse engineering. I preferred the BetrFS paper myself, and asked the authors to write for *;login:* about the B-epsilon trees used to speed up writes. But, not surprisingly, I wasn't part of the PC, which did the work of selecting the 21% of submitted papers.

The Test-of-Time award, based on papers published between 2002 and 2005 at FAST, went to "Hippodrome: Running Circles around Storage Administration," by Eric Anderson et al. Hippodrome is a tool used to automate the design and configuration process for storage systems using an iterative loop that continues until it finds a satisfactory solution to the target workload. Most of the authors, all HP Lab employees, were present to receive the award.

Erez Zadok wanted his students to have the experience of summarizing presentations. Erez also reviewed their summaries before the students sent them to me for editing.

The Theory of Everything: Scaling for Future Systems

Summarized by Ming Chen (mchen@cs.stonybrook.edu)

CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems

Alexander Thomson, Google; Daniel J. Abadi, Yale University

Alexander Thomson began by noting that their distributed file system was the first attempt to solve two difficult problems simultaneously: consistent file system replication over wide-area network (WAN), and scalable metadata management. He emphasized the importance of the problems and reviewed previous systems (HDFS, Ceph, etc.) which each addressed one of these problems but not both. He then went through a file creation example to demonstrate how CalvinFS handles file system operations and solves both problems.

CalvinFS achieves strong consistency across geographically distant replicas and high metadata scalability by using CalvinDB, a partitioned database system that supports fast distributed transactions. CalvinDB operates by (1) ordering and scheduling all transactions globally, (2) reading and broadcasting all transaction inputs, and then (3) deterministically committing the transactions on all replicas once all needed inputs and locks become available. CalvinDB's deterministic feature allows transactions to be committed without requiring a traditional two-phase commit protocol. Consequently, each CalvinDB transaction incurs only one network RTT latency instead of two as in two-phase commit. To be deterministic, the read- and write-set and locks of transactions need to be known in advance, which is not always possible: for example, when the write-set depends on the results of preceding read operations in the same transaction. CalvinDB solves this

using an Optimistic Lock Location Prediction (OLLP) mechanism. In addition to the metadata store based on CalvinDB, CalvinFS also has a variable-size immutable block store, which associates each block with a global unique ID (GUID) and assigns it to block servers using hashing.

CalvinFS can store 3 billion small files that are replicated among three geo-distributed datacenters (each running 100 EC2 instances) and could store even more if the authors could afford to run more instances. CalvinFS can handle hundreds of thousands of updates and millions of reads per second. By satisfying reads locally, CalvinFS has low read latency (median of 5 ms, and 99th percentile of 120 ms); however, as a tradeoff for strong global consistency, its writes suffer from WAN latency (200–800 ms).

In the Q&A session, Alexander acknowledged Zhe Zhang's (Cloudera) supposition that directory renaming is more expensive than other recursive operations, such as changing permission of a directory. Zhe then asked about the order between writing the block data and creating the file in the example. Alexander answered that the order does not matter as long as the two subtasks are committed at the same time. Brian (Johns Trading) wondered how CalvinFS handles conflicting operations made simultaneously by multiple datacenters. Alexander replied that conflicting operations are easy to detect and handle because CalvinFS has a global ordering of all operations.

Analysis of the ECMWF Storage Landscape

Matthias Grawinkel, Lars Nagel, Markus Masker, Federico Padua, and Andre Brinkmann, Johannes-Gutenberg University Mainz; Lennart Sorth, European Centre for Medium-Range Weather Forecasts

Matthias presented the first analysis of active archives (data to be accessed at any time), specifically a 14.8 PB general-purpose file archive (ECFS) and a 37.9 PB object database (MARS) at the European Centre for Medium-Range Weather Forecasts (ECMWF). Both systems use tapes as primary storage media and disks as cache media.

As of September 2014, the ECFS archive system had a 1:43 disk-to-tape ratio, and it contained 137.5 million files and 5.5 million directories. From 1/1/2012 to 5/20/2014, ECFS served 78.3 million PUT requests (11.83 PB in total size) involving 66.2 million files; 38.5 million GET requests (7.24 PB in total size) involving 12.2 million files; 4.2 million DELETE requests; and 6.4 million RENAME requests. The requests demonstrated high locality, enabling an 86.7% disk cache hit ratio. Nevertheless, 73.7% of the files on tape have never been read. In the simulation study of various caching algorithms, the Belady algorithm performed the best, the ARC and LRU algorithms followed closely after, and MRU was the worst.

The MARS object database had a 1:38 disk-to-tape ratio, and contained 170 billion fields in 9.7 million files and 0.56 million directories. From 1/1/2010 to 2/27/2014, MARS served 115 million archive requests and 1.2 billion retrieve requests. MARS's cache was very effective, and only 2.2% of the requests needed to read data from tapes. Similar to ECFS, 80.4% tape files in MARS were never read.

Matthias also presented some interesting tape statistics in the HPSS backing ECFS and MARS: (1) there were about nine tape loads per minute; (2) about 20% of all tapes accounted for 80% of all mounts; (3) the median of the tape loading time was only 35 seconds, but its 95% and 99% percentiles were two and four minutes, respectively.

Matthias concluded that disk caches on top of tapes are efficient in non-interactive systems. He noted two drawbacks of heavy use of tapes: high wear-out and unpredictable stacking latencies. He also expressed one concern, considering that the decrease in per-bit storage cost is slowing, of accommodating the fast-growing storage requirement (up to 53% annually) under a constant budget. Their traces and scripts are published at https://github.com/zdvresearch/fast15-paper-extras in order to help build better archive systems.

Umesh Maheshwari (Nimble Storage) wondered what the cost effect would be if the data were stored purely in disks since disk price had dropped and disks could eliminate the difficulties of loading tapes. Matthias thought pure disks would still be more expensive, especially considering that disks keep spinning (and consuming energy). Brent Welch (Google) commented that it should be possible to simulate how much more disk cache would be needed for the tape-and-disk hybrid system to perform as well as the more expensive pure-disk solution. John Kaitschuck (Seagate) wondered whether the data reported in this study contained traffic introduced by data migration (e.g., from an old format to a new format). Matthias was not sure of the answer.

Efficient Intra-Operating System Protection against Harmful DMAs

Moshe Malka, Nadav Amit, and Dan Tsafrir, Technion–Israel Institute of Technology

Moshe presented their work on improving the efficiency of using IOMMU (Input/Output Memory Management Unit), whose relationship to I/O devices is similar to the regular MMU's relationship to processes. IOMMU manages direct memory accesses (DMA) from I/O bus to main memory by maintaining mappings from I/O virtual addresses to physical memory addresses. Similar to MMU's TLB cache, there is an IOTLB cache for IOMMU. To protect the OS from errant/malicious devices and buggy drivers, IOMMU mappings should be established only for the moment they are used and be invalidated immediately after the DMA transfers. However, the IOTLB invalidation was slow and the IOVA (I/O Virtual Address) subsystem opts for batching (deferring) multiple invalidations.

Using 40 Gbps NICs as examples, Moshe showed from benchmarking data that the slowness of the invalidation was actually mainly contributed by the IOVA allocator software instead of the IOTLB hardware. The IOVA allocator used a constant-complexity algorithm based on a red-black tree and a heuristic.

The allocator works well if the NIC has one transfer ring and one receive ring. However, modern NICs usually have many transfer and receive rings, interaction among which can break the heuristic and convert the algorithm's complexity to linear. To solve the problem, Moshe showed how they converted the algorithm back to constant-complexity by simply adding a free list before the red-black tree. Through comprehensive evaluations, Moshe showed they improved the IOVA allocator by orders of magnitude and improved workloads' overall performance by up to 5.5x. They performed their study on Linux, but they found a similar problem on FreeBSD.

Raju Rangaswami (FIU) asked whether they had found similar problems in fast multi-queue SSDs, which likely suffer from the same interference among different queues. Moshe replied that another student in his lab was actively working on that and might come up with results in a few months.

Big: Big Systems

Summarized by Akhilesh Chaganti (akhilesh.chaganti@stonybrook.edu)

FlashGraph: Processing Billion-Node Graphs on an Array of Commodity SSDs

Da Zheng, Disa Mhembere, Randal Burns, Joshua Vogelstein, Carey E. Priebe, and Alexander S. Szalay, Johns Hopkins University

Da Zheng started the Big Systems session with their paper on large-scale graph analysis. He introduced FlashGraph, a semi-external memory-based graph analysis framework built over commodity flash arrays to achieve scalability and performance comparable to in-memory alternatives. He cited the application of graphs in many real world problems ubiquitously across industry and research as the main driver for developing cost-effective efficient alternatives like FlashGraph. Da Zheng explained the major challenges involved in graph analysis: (1) massive sizes with billions of nodes and edges, (2) the randomness of vertex connection, and (3) the power-law distribution in vertex degree, in which only a small number of vertices connect to many other vertices, making load balancing a problem in distributed environments. Da Zheng then presented the audience with the some popular alternatives employed in graph analysis and their problems. The first option is to use expensive machines with large RAM where the problem of cost and scalability is evident. Another alternative is to leverage a cluster to scale out graphs. Although this is popular, it is plagued by network latencies owing to the random vertex connections and the frequent network communications they invoke. The third option is to scale graph analysis using HDDs thanks to its capacity to store very large graphs. But random I/O performance and frequent I/O from external memory-based graph engines makes this solution very slow.

Da Zheng presented his position on how SSDs could offer a better solution in scaling graphs. SSDs have some obvious advantages compared to HDDs in terms of throughput and latency, while being cheaper and larger compared to RAM. But there are some potential problems that need to be addressed before using SSDs: heavy locking overhead for reads on large SSD arrays, and low throughput and high latency compared to RAM. Three design decisions are used to tackle these problems: reducing I/O, overlapping I/O and computation, and sequential I/O. All the desired choices are implemented using SAFS, a user space file system optimized for SSD array. Da Zheng then talked about the rest of the design, where FlashGraph communicates with SAFS and provides a graph-programming interface to the application layer. The main task of FlashGraph is to schedule the vertex programs written by the users and to optimize the I/O issues by those programs. When FlashGraph receives tasks, part of the computation is pushed to page cache to overlap computation and I/O and to reduce the memory overhead. The FlashGraph solution maintains the vertex state of the graph in memory while storing the edge lists on SSDs. The advantages with this model are quite evident. Compared to significant network-based communications in the distributed memory model, this model provides in-memory communication for all the vertices. This is better than the external memory model in terms of I/O. Even with SSDs, there is need for heavy I/O optimization. To achieve this, FlashGraph fetches only the lists required by the application. It also conservatively merges I/O requests on same or adjacent pages to increase sequential I/O.

Da Zheng presented the performance results of FlashGraph over some frequently used access patterns of graph applications. The classes of applications consisted of (1) those where only a subset of vertices access their own edge lists (e.g., Breadth First Search); (2) applications where every vertex accesses their own edge list (e.g., PageRank); and (3) applications where vertices access edge lists of other vertices (e.g., Triangle Count). The authors compared FlashGraph with other in-memory alternatives like PowerGraph (distributed memory) and Galois; to better understand its performance, they also benchmarked it against an in-memory implementation of FlashGraph. The in-memory implementation of FlashGraph is comparable to the performance of Galois, and the semi-external version of it is comparable to in-memory FlashGraph, sometimes reaching 80% of its performance. Surprisingly, FlashGraph outperforms PowerGraph in most cases. FlashGraph is also benchmarked for its scalability using the publicly available largest graph with 3.5 billion nodes and 129 billion edges. All classes of the algorithms run in a reasonable amount of time (from five minutes for BFS to 130 minutes for Triangle Count) and space (from 22 GB for BFS to 81 GB for Betweenness Centrality). The authors also used runtime results of Google Pregel and Microsoft Trinity on a smaller problem and better infrastructure to indirectly compare and conclude that FlashGraph outperforms them. Da Zheng concluded his talk saying that FlashGraph has performance comparable to in-memory counterparts and has provided opportunities to perform massive graph analysis on commodity hardware.

In the Q&A session, one participant asked why not use a NVME interface instead of using different file systems to improve read

performance of SATA-based SSD and wondered whether reads would be compute-bound irrespective of the interface. Da Zheng replied that NVME will definitely improve performance because SAFS merges the I/O, which makes large and sequential reads overlap computation with I/O. Brad Morrey (HP Labs) asked whether adjacent pages in SSD were physically adjacent. Da Zheng replied they use the mapping table in SSD.

Host-Side Filesystem Journaling for Durable Shared Storage

Andromachi Hatzieleftheriou and Stergios V. Anastasiadis, University of Ioannina

Andromachi Hatzieleftheriou gave a lively presentation on improving the durability of shared data storage in a distributed file system environment in a typical datacenter-like clustered infrastructure. Before talking about durability problems, the author first outlined how typical datacenter storage is implemented as a multi-tier distributed system across clustered commodity servers and storage. In such a system, data is usually replicated on the client-side followed by a caching layer and the backend storage. The clients are implemented as stateless entities to reduce the communication across layers, which compromised the durability guarantees on the data present in the client's volatile memory in case of crash/reboot. The author has demonstrated the level of inconsistency such a model can cause over the Ceph object-based scale-out file system. While running Filebench fileserver workload over ten thousand files for two minutes, on average 24.3 MB of dirty data is present in the volatile memory because of write-back happening every five seconds over data older than 30 seconds. Network storage is usually implemented by providing one of either file-based, block-based, or object-based interfaces to clients. One important aspect of shared storage is the durable caching on the client side for performance and efficiency. Most approaches use block-based caching, which comes with significant problems like file sharing, block translation overhead, and consistency issues that pop up due to the semantic gap between file and block layers, which complicates the atomic grouping of dirty pages and the ordering of I/O requests.

The author then described how they improved durability of in-memory based caches on the host side using a file-based interface with better performance and efficiency. Key principles of their proposed storage architecture included a POSIX-like file interface with file sharing across different hosts, durability of recent writes in case of client crash/reboot, improved performance for write requests, and efficiently scaling out the backend storage. The author proposed Arion which uses these design principles and which relies on scale-out object-based backend system with multiple data (DS) and metadata servers (MDS). The clients (bare-metal or virtual) of the distributed file system are integrated with a disk-based journal that logs both data and metadata of I/O requests before the predefined numbers of backend replicas are created. Dirty data in memory is synchronously transferred to the journal either periodically or on explicit flush, and write-back to the backend server is initiated periodically or under tighter space conditions of memory or journal. Once write-back is complete, the corresponding entries in the journal are invalidated. The author then talked about handling the consistency issues during file access. Shared file access is achieved through the tokens leased to the client by MDS. Upon any concurrent conflicting operation by a different client, the client first checkpoints the pending updates, which is followed by invalidation of journal entries before revocation of token. Upon client reconnection or reboot, the client acquires the token again and replays file updates if journaled metadata is newer than metadata fetched from backend storage. For implementation, the Linux JBD2 is used to manage the journal, which is integrated with CephFS kernel-level client. In order to manage the file metadata in the journal, the JBD tag is expanded with fields to identify the metadata. This tag is used to compare the changes with metadata fetched from MDS.

The author provided the performance results of various experiments conducted on virtual clients over the Linux host with a storage backend cluster of five machines running Arion and Ceph. Arion was allocated a 2 GB partition for the local journal. All the experiments were based on Filebench and FIO workloads. Different configurations of Ceph and Arion were included in the benchmarks. Ceph was examined with write-back and expiration times set to 5 and 30 seconds (Ceph) and with both times set to 1 second (Ceph-1). Another configuration of Ceph was used where the file system was synchronously mounted (Ceph-sync) eliminating caching. Arion was set to copy changes to the journal every 1 second, while the write-back and expiration times were set to 60 seconds (Arion-60) or infinity (Arion-inf). The benchmarks were performed to analyze performance, durability, and efficiency of Arion compared to the Ceph kernel level client. Arion was found to reduce the average amount vulnerable data in memory to 5.4 MB for Filebench fileserver workload, significantly outperforming the Ceph client (24.3 MB). In another experiment over the Filebench Mailserver workload, Arion achieved up to 58% higher throughput than Ceph. Network traffic was also reduced by 30% in received load and 27% in transmitted load at one of the OSDs of the Ceph backend. Both the systems were also explored with the FIO microbenchmark with Zipfian random writes. The Arion-60 configuration achieved 22% reduced write latency compared to the default Ceph configuration, and total network traffic was reduced by 42% at one OSD. In the same experiment, the authors benchmarked bandwidth utilization of the file system of OSD and interestingly found that Arion reduced total file system disk utilization by 82%. Andromachi concluded her talk by giving the direction to future work and reiterated the benefits of host-side file journaling.

In Q&A, one participant wondered why the Ceph-sync configuration, where the file system is mounted with sync, had better throughput and network load. Another participant applauded the work and wondered how Arion handles a data loss case

where the client commits to the local journal and goes down and meanwhile another client updates the corresponding file in the backend storage.

LADS: Optimizing Data Transfers Using Layout-Aware Data Scheduling

Youngjae Kim, Scott Atchley, Geoffroy R. Vallee, and Galen M. Shipman, Oak Ridge National Laboratory

Youngjae Kim began by giving the background of big data trends across scientific domains. The major challenges are storing, retrieving, and managing such extreme-scale data. He estimated that the Department of Energy's (DOE) data generation rates could reach 1 exabyte per year by 2018. Also, most scientific big data applications need data coupling, i.e., combining data sets physically stored in different facilities. So it is evident that efficient movement of such massive data is not trivial. Youngjae also noted that DOE is planning to enhance their network technology to support a 1 TB/s transfer rate in the near future. But this does not solve the problem because data is still stored on slow devices.

To motivate the research around finding efficient data movement solutions, Youngjae talked about data management at Oak Ridge Laboratory Computing Facility, which runs Titan, the fastest supercomputer in the US. The spider file system based on Lustre provides the petascale PFS for the multiple clusters including Titan. Concurrent data accesses from clusters lead to contentions across multiple network, file system, and storage layers. In such an environment, data movement is usually performed through data transfer nodes (DTN), which access the PFS and send data over the network to the destination's DTN. Because of contentions, PFS can become bottlenecked during data transfers across DTNs. The main problem here is that the difference between the network bandwidth and I/O bandwidth is growing, and PFS's I/O bandwidth is underutilized in the existing schemes of data transfers.

To improve the utilization of PFS's bandwidth on DTN for big data transfers, Youngjae introduced Layout Aware Data Scheduling (LADS) as a solution. He first explained the problem with traditional approaches and used Lustre as the base PFS. He described Lustre's architecture, which contains a metadata server (MDS) and an object storage server (OSS). MDS holds the directory tree and stores metadata about files and does not involve file I/O. On the other hand, OSS manages object storage targets (OSTs), which hold the stripes of file contents and maintains locking of file contents when requested by Lustre clients. In a typical scenario, the client requests file information from MDS, which returns the location of the OSTs holding the file. The client then makes RPC connections to file servers to perform I/O. Youngjae made some key observations about Lustre PFS: it views the file system as single namespace. But the storage is not on a single disk—rather, storage is on multiple disk arrays on multiple servers. PFS is also designed for parallel I/O and traditional data transfer protocols and does not fully utilize inherent parallelism. The main reason behind this is that the traditional file-based approach completely ignores the layout information of the file. For instance, if two threads are working on different files present over the same OST or disk, the threads contend for OST access. In such cases, the thread application runtime increases by 25%. In another case, when multiple threads work on a single file, the parallelism is limited by stripe sizes across OSTs. Multiple threads can contend for different stripes of the same file on the same OST, increasing the runtimes by 50% in some cases. Existing toolkits for bulk data movement (GridFTP, bbcp, RFTP, and SCP) suffer poor performance because of this logical view of files.

LADS uses a physical view of files with which it can understand physical layout of the objects of the file, the storage target holding the objects, and the topology of storage servers and targets. With this knowledge a thread can be scheduled to work on any object on any OST so that we can avoid contention that could have happened in earlier cases. LADS has four design goals: (1) maximized parallelism on multicore CPUs, (2) portability for modern network technologies (using Common Communication Interface (CCI)), (3) leverage parallelism of PFS, and (4) improved hotspot/congestion avoidance, which leads to an end-to-end data transfer optimization reducing the difference between faster network and slower storage. Youngjae then gave a lucid description of the LADS architecture. Data transfers happen between two entities called LADS source and sink. Each of them implements CCI for communication and creates RMA buffers on DRAM. It implements three types of threads: master thread to maintain transfer state, I/O thread for reading/writing data chunks, and comm thread for data transfer across DTNs. Moreover, LADS implements layout-aware, congestion-aware, and NVRAM-buffering algorithms. LADS implements as many OST queues as there are OSTs. A round robin scheduler removes the data chunk from these queues and places it in an RMA buffer while it's not full. When it's full, the I/O threads are put to sleep while the comm thread transfers the contents of the RMA buffer. Once the RMA is available, the I/O threads carry on the leftover I/O and place the chunks in respective queues. I/O congestion control is implemented using a threshold-based reactive algorithm. A threshold value is used to determine the congested servers, and a skip value is used to skip a number of servers. An NVM buffer is also used as extended memory if the RMA is full, which is a typical scenario when the sink is experiencing widespread congestion.

With this overview, Youngjae presented the performance results of LADS. Two Intel Xeon servers were used as DTNs in the testbed and connected with two Fusion-I/O SSDs for NVM and were backed by a Lustre file system over 32 HDDs. An IB QDR 40G network connected the two DTNs. To fairly evaluate the framework, the author increased the message size to make sure the storage bandwidth was not over-provisioned compared to network bandwidth. At 16 KB, the maximum network bandwidth was 3.2 GB/s whereas the I/O bandwidth was 2.3 GB/s. A snap-

shot of the spider file system revealed that 85% of files were less than 1 MB, and the larger blocks occupied most of the file system space. So for benchmarking, one hundred 1 GB files (big files workload) and ten thousand 1 MB files (small files workload) were used. For the baseline, in the environment without congestion, the transfer rate increased linearly with the number of I/O threads for both workloads, whereas traditional bbcp does not improve with an increase in the number of TCP streams to operate on the same file. This experiment also revealed that LADS moderately uses CPU and memory.

LADS was also evaluated on a storage-congested environment. To simulate congestion, a Linux I/O load generator was used. When a source was congested, a congestion awareness algorithm performed up to 35% better than the baseline configuration without congestion awareness. When sink was congested, the performance impact was significant compared to that of the source congestion. Due to time constraints, the author summarized the remaining experiments. He evaluated the effectiveness of the NVM buffer at source. Throughputs increased with an increase in the size of the available buffer. Actual DTNs at ORNL were also evaluated by transferring data from one cluster with a 20 PB file system to another cluster; LADs showed 6.8 times higher performance compared to bbcp. Youngjae summarized this section and reiterated the effectiveness of layout-aware and congestion-aware models in efficient data transfers. He concluded the talk with his vision to have an optimized virtual path for data transfer from any source to any sink so as to promote collaborative research among organizations.

In the Q&A session, one participant asked the author to compare and contrast the Lustre community's Network Request Scheduler with LADS. Youngjae pointed out that with knowledge of file layout, many smart scheduling schemes could be employed at the application level. The questioner pointed out that the perfectly working environment used in the evaluation does not exist in production environments like that of ORNL, where events like disk failures and Lustre client evictions are common. The questioner wondered whether the authors considered such scenarios in designing LADS. Youngjae mentioned that the current work does not deal with failures.

Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-Bandwidth

K.V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran, University of California, Berkeley

K.V. Rashmi began by presenting background for using redundancy in distributed storage environments. Historically, durability and availability of data became essential elements in the design of distributed storage. One of the popular ways to achieve this redundancy is to replicate data across multiple locations. An alternative approach is to employ erasure code to achieve this. It is well known that erasure codes utilize the storage space more efficiently to achieve redundancy when compared to replication. Traditional codes like Reed Solomon provide the maximum possible fault tolerance for the storage overhead used. But the more interesting metric in evaluation of erasure codes is the maintenance costs for the reconstruction of lost data in order to maintain the required level of redundancy. This is quite a frequent operation in distributed systems and becomes heavy in terms of network and I/O. Traditional erasure codes are particularly inefficient in this respect. Both the theory and systems research communities have been working on this problem for quite some time. As a result, a powerful class of erasure coding framework is proposed that optimizes storage and network bandwidth costs of reconstruction. Rashmi extended this class of coding techniques to optimize for I/O as well.

Before delving into the details of this work, Rashmi gave an overview on why traditional codes are inefficient for reconstruction. In a replicated environment, to reconstruct a lost block, one block from a different location has to be transferred, which incurs the cost of I/O and network in transferring the block. In general, one of the popular codes like Reed Solomon takes in k data blocks and generates $n - k$ parity blocks using carefully designed functions for optimal storage and fault tolerance. Now any of the k out of n blocks can be used to reconstruct the missing data. It is clearly evident that the I/O and network costs bump up k times compared to replication cost; for typical variables this cost is increased 10–20x. To address this problem, one of the effective solutions is to use Minimum Storage Regeneration (MSR) codes, which optimize storage and network bandwidth by figuring out the minimum amount of information needed to transfer for reconstruction. The MSR framework, like Reed Solomon, has k data blocks and $n - k$ parity blocks. Under MSR, any block can be reconstructed by connecting to any d ($> k$) helper blocks of remaining blocks and by transferring a small amount of data from each. The total amount of data transferred is significantly smaller compared to Reed Solomon. Although MSR optimizes storage and network bandwidth, it does not consider I/O optimization. In fact I/O is much worse compared to Reed Solomon because the nodes performing reconstruction read the entire block and transfer only a small amount of information. This optimizes bandwidth but performs d full block reads (I/O) whereas Reed Solomon performs only k block reads. In summary, the MSR framework optimizes storage and network but performs much worse in I/O. This work attempts to improve the I/O while retaining the storage and network bandwidth optimization.

Rashmi presented two algorithms that transform MSR codes into codes that provide efficient I/O, storage, and network. The first algorithm transforms MSR codes so that they locally optimize I/O at each of the helper blocks, while the second algorithm builds on top of first to minimize I/O costs globally across all blocks. Rashmi first discussed the kind of performance that can be obtained by applying the transformations. The transformations are applied to a class of practical MSR codes called Product Matrix MSR (PM) codes. PM codes work with storage overhead of less than 2x and provide optimal fault tolerance. They are

usually employed in applications that need high fault tolerance. The author implemented both the original PM codes and transformed (PM_RBT) codes in C and evaluated them on an Amazon EC2 cluster, employing Jerasure2 and GF-complete for implementing finite field arithmetic and Reed Solomon. The evaluation was performed for six data blocks with five parity blocks and a block size of 16 MB. The author observed that both the original PM and PM_RBT had approximately 3.27x less data transferred compared to Reed Solomon and emphasized that transformed code retained the bandwidth optimality of the original PM codes. In terms of IOPS consumed during reconstruction, it was evident that PM codes required more IOPS than Reed Solomon. PM_RBT saved up to 5x fewer IOPS than PM and 3x fewer IOPS than Reed Solomon, showing significant performance improvement in terms of I/O. A similar trend was observed for the higher block size of 128 MB. Rashmi next presented I/O completion time for reconstruction. The transformed code (PM_RBT) resulted in 5 to 6 times faster I/O making it significantly faster than Reed Solomon and PM codes.

With performance results in mind, Rashmi described how the transformations work using two algorithms. In MSR, a helper unnecessarily reads a whole block even though it only needs a minor portion of the block; optimally, it reads just the information that's needed. When a helper works in this approach, it's called "reconstruction-by-transfer" (RBT), i.e., it does not do any computation but just reads and transfers. The first algorithm transforms MSR codes to achieve RBT to the maximum extent possible and is applicable to all MSR codes with two properties. First, the function computed at the helper is not dependent on blocks from other helpers (i.e., each block has a predetermined function that aids in reconstruction of another block). The next property deals with independence of these functions at a particular block. Any subset of functions, which produces data of block size, is considered to be independent. The main idea behind this algorithm is to pre-compute and store the results of one such independent set in a block. So now this block can be used for the reconstruction of the blocks corresponding to the functions in the subset. Under MSR, whenever a helper does RBT, it can simply read the data corresponding to a particular function in the block, and hence only a minimum amount of data is read and transferred. But the question of how to choose which functions for which block is dealt with by the second algorithm. Rashmi mentioned that it uses a greedy approach to optimally assign RBT-helpers to minimize I/O cost globally across all the blocks and asked the users to refer to the paper for detailed information on how it's done. She discussed two extreme cases of this algorithm: (1) all the blocks helping all the data blocks to the maximum extent possible (SYS pattern) and (2) all the blocks getting equal treatment and each block helping the following blocks in a cyclic fashion (CYC pattern). Rashmi also evaluated and concluded that the RBT does not affect the decoding speed of PM and is similar to that of Reed Solomon. The encoding speed is slower than that of Reed Solomon but is still practical; she also observed that RBT-SYS has a higher encoding speed than PM. With this, Rashmi summarized the work and restated the benefits brought in by these transformations.

In the Q&A session, one participant was curious about how the compute bandwidth was traded off, because with more helpers the speed is retained but the equations are more complex. Rashmi agreed that there is a compute tradeoff in resource utilization optimization, but it is still practical enough. She reminded the audience that the compute cost during decoding was similar to the cost of RS decoding with two parities. The compute cost is actually not too high because the amount of data churned is much less compared to Reed Solomon. Keith Smith (NetApp) pointed out that the presentation examples used one erasure and wondered about how the efficiency would turn out when there are multiple erasures. Rashmi replied that currently the MSR framework considers optimizing for a single failure and reiterated since only *d* helpers are needed for reconstruction there is room for considering multiple failures in that respect. She also noted that most of the typical scenarios are single failure.

The Fault in Our Stars: Reliability

Summarized by Zhen Cao (zhccao@cs.stonybrook.edu)

Failure-Atomic Updates of Application Data in a Linux File System

Rajat Verma and Anton Ajay Mendez, Hewlett-Packard; Stan Park, Hewlett-Packard Labs; Sandya Mannarswamy, Hewlett-Packard; Terence Kelly and Charles B. Morrey III, Hewlett-Packard Labs

Anton Ajay Mendez presented this paper on behalf of his colleagues in HP Labs, which mainly introduces the design, implementation, and evaluation of failure-atomic application data updates in HP's Advanced File System (AdvFS). Failure-atomic updates would want applications to make their data structures persistent and consistent. Applications would maintain a series of atomic sync points, which provides the ability to revert back to the previous successful sync point in the event of failures. Ajay compared their work with existing mechanisms, including relational databases and key-value stores, and listed some of the previous work and their limitations. Inspired by failure-atomic Msync, their work provides a better and more generalized solution than Msync.

Ajay then came to their solution, which is called O_ATOMIC. It is a flag that can be passed to open() system call, and every subsequent sync would make intervening writes atomic. They also provided another call, syncv, for failure-atomically modifying multiple files. The detailed implementation of their solution leverages a file clone feature, which is a writable snapshot of the file. When a file is opened with O_ATOMIC, a clone of the file is made. When the file is modified the changed blocks are remapped via COW mechanism, while the clone still points to the original blocks. When a sync is called, the old clone is deleted and a new clone is created. Recovery happens lazily on lockup: if a clone exists, delete the file and rename the clone to the original

file. Ajay said that their mechanism can be implemented in any file system that supports clones.

To verify the correctness of the mechanism, they injected two types of failure: crash point tests and power cycle tests. Ajay said that test results showed that no corruption happened when files opened with O_ATOMIC. He compared the performance of their implementation against existing key-values stores, and proved it is efficient enough. He also mentioned some caveats of their implementations. Clones introduce fragmentation, but online defragmentation may help here. Multi-process file updates need to be carefully coordinated by the application.

During the questions, Haryadi Gunawi (University of Chicago) asked whether they evaluated how much fragmentation the deployment of their implementation introduced. Ajay replied that they haven't done evaluation on that yet. Justin Paluska (EditShare) asked why multi-process coordination is complicated. Ajay answered because you need to take care of the concurrency issue. What's more, if multiple processes opened the same file, and one process crashed, there is no way to detect it. John Ousterhout (Stanford University) asked about the performance implications of this mechanism, especially when cloning large files. Ajay said if you do the sync immediately after the modification, there is an additional overhead, and it is related to the amount of data changed.

A Tale of Two Erasure Codes in HDFS

Mingyuan Xia, McGill University; Mohit Saxena, Mario Blaum, and David A. Pease, IBM Research Almaden

Mingyuan Xia gave a lively presentation of their new erasure-coded file system, Hadoop Adaptively-Coded Distributed File System (HACFS). He began by showing the fast-growing trend toward global data and laying out the timeline of distributed storage systems. Because of high storage overhead, most systems began using erasure coding instead of the original replication-based methods. Mingyuan gave an overview of erasure coding, using the Reed-Solomon code in Facebook's HDFS as an example. However, erasure coding brings the problems of high degraded read latency and longer reconstruction time. He further explained the two problems in detail. Their goal in this work is to design a technique that can achieve faster recovery as well as lower storage overhead.

Mingyuan then presented the HDFS data access skew. Ten percent of the data gets the majority of the accesses, and 90% of the data is accessed only a few times. Motivated by this finding, they coded read hot files with a fast code with low recovery cost, and coded the majority of the data with a compact code with high storage efficiency. He used the product code family as an example to illustrate this idea. The system will adapt to the workload by converting files between fast and compact codes. Mingyuan described the details of this conversion. When total storage overhead exceeds the storage bound, the system will select files encoded with fast code and upcode them to compact code. Similarly but less likely, when the system is way below the bound, files would be chosen to be downcoded. He provided an example of upcoding operation of the product code family.

Mingyuan then presented the details of environment setting and workloads for evaluation of HACFS. He also formally defined their evaluation metrics: they are degraded read latency, reconstruction time, and storage overhead. He compared HACFS with other distributed storage systems as well as with other erasure coding mechanisms. Mingyuan showed that HACFS always maintains a low storage overhead, while improving the degraded read latency, reconstruction time, and network and disk traffic to a certain extent.

During the questions, Brent Welch (Google) asked whether the codes being compared all have the same redundancy. Mingyuan answered yes. Welch further asked about the data trace, because if we have too much cold data, the recovery of cold data would dominate. Mingyuan answered that in the HDFS data access skew, they showed that the majority of the files are created, read only once, and then stay there. Another researcher asked whether we will lose redundancy by using erasure codes. Mingyuan replied that in three-way replication, replicas are supposed to be placed in different nodes, and it is the same case with erasure codes. So when a single machine fails, no two blocks in one strip will lose at the same time. Rashmi Vinayak (UC Berkeley) asked whether reliability decreases after converting from fast codes to compact codes. Mingyuan answered that they have shown that it is no worse than the three-way replication and is comparable to LRC codes.

How Much Can Data Compressibility Help to Improve NAND Flash Memory Lifetime?

Jiangpeng Li, Kai Zhao, and Xuebin Zhang, Rensselaer Polytechnic Institute; Jun Ma, Shanghai Jiao Tong University; Ming Zhao, Florida International University; Tong Zhang, Rensselaer Polytechnic Institute

Jiangpeng Li began by giving an overview of NAND flash memory. He first explained how NAND flash memory cells gradually wear out with program/erase (P/E) cycling. Methods have been proposed to improve the lifetime of flash memories, including log-structured file system, flash translation layer, error correction coding, and data compression.

Jiangpeng claimed that because of unused space in one NAND flash page and the impact of compression ratio variance, the common sense perception of the quantitative relationship between data compressibility and memory lifetime improvement will not always hold. Moreover, NAND flash memory may further experience content-dependent memory damage. He then showed test results on raw bit error rates (BER) of four different patterns of content to support this. Inspired by this finding, they fill unused space with certain bits so that the number of low-damage patterns is increased and high-damage patterns are reduced. Jiangpeng then proposed implicit data compression as an alternative to complement explicit data compression.

Jiangpeng introduced the damage factor to quantify the impact of different content on memory cell damage. He then derived the mathematical model needed for estimating flash memory lifetime. Simulation results on storage device survival probability when storing different types of files showed the lifetime improved through use of data compression storage techniques. He also discussed the impact of different data compression ratio means and variances on the lifetime gain.

During questions, Nirmal Saxena (Samsung) asked whether the sensitivity in lifetime would change if the wear-leveling algorithm was introduced. Jiangpeng answered that their model can support various kinds of data types. Another researcher asked about the effect of one compressed block being placed on two blocks on the lifetime of flash memory. Jiangpeng replied that in this work they assumed that data are compressed by data sectors. Justin Mazzola Paluska (EditShare) asked how using encrypted file systems would affect the results of this work. Jiangpeng answered that data compression actually would complicate the file system, and in this work they just assumed the granularity of compression is a data sector.

RAIDShield: Characterizing, Monitoring, and Proactively Protecting against Disk Failures

Ao Ma, Fred Douglis, Guanlin Lu, and Darren Sawyer, EMC Corporation; Surendar Chandra and Windsor Hsu, Datrium, Inc.

ACM Test-of-Time Fast Track Award!

Ao Ma gave a lively presentation on disk failure analysis and proactive protection. Ao began by showing that disk failures are commonplace and by giving an overview of RAID. Adding extra redundancy ensures data reliability at the cost of storage efficiency. By analyzing the data collected from one million SATA disks, they revealed that disk failure is predictable, and built RAIDShield, an active defense mechanism, which would reconstruct failing disks before it becomes too late.

Ao then gave the formal definition of a whole-disk failure and showed the statistics summary of their data collection. Disk failure distribution analysis showed that a large fraction of failed drives fail at a similar age. Also, the number of affected disks with sector errors keeps growing, and sector error numbers increase continuously. As a result, passive redundancy is inefficient. Instead, RAIDShield would proactively recognize impending failures and migrate vulnerable data in advance. Using experimental results, Ao showed that reallocated sector (RS) count is a good indicator for failures, while media error count is not. He then characterized its relation with disk failure rate and disk failure time.

Based on previous findings, their single disk proactive protection, called PLATE, uses RS count to predict impending disk failure in advance. Experiment results showed that both the predicted failure and false positive rates decrease as the RS count threshold increases. Ao analyzed the effects of PLATE by comparing the causes of recovery incidents with and without proactive protection, and found that RAID failures were reduced by about 70%. However, PLATE may miss RAID failures caused by multiple less reliable drives, which motivated them to introduce ARMOR, the RAID group proactive protection. Ao then gave an example of how disk group protection works. It calculates the probability of a single disk failure and a vulnerable RAID. Evaluation results also show that ARMOR is an effective methodology to recognize endangered disk groups. In the end, Ao went through some of the related work.

During the questions, John Ousterhout (Stanford University) asked whether the distribution of lifetime of failed drives were percentages of total disks or only of failed disks. Ao answered they are the percentages of failed drives. John followed up about the peak time of failure. Ao replied that they are still not sure why the peak time happens around the third year. Another researcher asked whether all the disks ran for the same span of time. Ao answered yes. They wouldn't replace a disk if there were no errors. A questioner wondered about the zero percentage of failures in the first year. Ao replied that their paper covered six different types of drives, but his presentation only examined the failure patterns of two types. Arkady Kanevsky (Dell) asked whether their model would hold for disks of different types and manufactures and whether their model indicates that we could step back from more complicated protection methods into simpler models. For the first question, Ao replied that now they only considered SATA drives and that it would be interesting to try other types of disks. Regarding the second question, Ao said that they are still polishing their monitor mechanism.

2015 USENIX Research in Linux File and Storage Technologies Summit

February 19, 2015, Santa Clara, CA

Summarized by Rik Farrow

Christoph Hellwig, a Linux I/O developer, opened the workshop by having participants introduce themselves. There were professors and students from Florida International University (FIU), Carnegie Mellon (CMU), Stony Brook University (SBU), Kookmin University (KU), and the University of Wisconsin (UW) present, as well as people from Google, EMC, Red Hat, Parallels, and IBM Research. By the end of the afternoon, one proposal would gain enthusiastic acceptance, while the person who had a proposal accepted in 2014 updated us on his progress.

Attendees had submitted six proposals for discussion, four of them based on FAST '15 papers. Christoph wanted to start working through the proposals immediately, but Erez Zadok asked whether there could first be some discussion of how the Linux kernel submissions process works.

Christoph mentioned that you can use the various kernel mailing lists (see [1] for names), and then displayed a nice diagram

for the Linux storage stack [2]. Christoph mentioned different list names, as he pointed to different blocks on the diagram. He used the linux-scsi list most often, but there are other lists, with the kernel list being the most useless because of the amount of list traffic.

Greg Ganger (CMU) asked what Christoph was using for his Non-Volatile Memory (NVM) development. Christoph said that he has an emulator that allows him to play with NVM. Erez pointed out that there is no single way to use NVM, and Christoph responded that some vendor has been using NVM for a while. One way of handling NVM is to mmap() sections of it into user memory. Erez then asked where NVM would fit into the diagram, and Christoph said it was modeled on the direct I/O code (DIO) alongside the Virtual File Systems (VFS) box.

Raju Rangaswami (FIU) asked how the I/O path would be laid out for NVDIMMs (NVM on the memory bus as opposed to PCIe), with the answer also being DIO. James Bottomley (Parallels) said that the diagram is a bit misleading, as all block I/O goes through the block cache, but DIO has its own structure.

Don Porter (SBU) asked about the politics surrounding DIO. Christoph replied that Linus doesn't like it. DIO is the generalization of the classic UNIX raw device. Raw devices bypass the block cache, and you can set a flag turning off the block cache for a device. Database developers and vendors, like Oracle, as well as virtualization developers want to manage block caches for themselves, which is where the interest in raw devices comes from.

Raju moved the discussion back to NVM, asking about Linux developers' view on the future of persistent memory. Christoph answered that what matters is the implementation but not the semantics. Right now, there are two cases: NVM that works like DRAM, and NVM that sits on the PCIe bus. If NVM works like DRAM, it gets treated like DRAM. James pointed out that NVM is not identical to DRAM, because it has slower writes. He also said that we don't have a view—we are looking for a good implementation. Intel has had some interesting failures in this area, and implementation for NVM appears to be an iterative process. Christoph retorted that some NVM is battery-backed DRAM, so it may be identical.

Erez asked about support for Shingled Magnetic Recording (SMR) and zone-based storage devices. Christoph replied that vendors have been pushing SMR on the kernel developers very hard, and they don't like that. James mentioned that they don't have pure SMR drives yet, because if used improperly, the drives' performance would be bad. Christoph described that scenario as host-managed drives, where you manage the drive yourself, and that host-managed SMR will not be in the kernel soon. The kernel can detect these drives, and use the SCSI path to support them, but you need your own code to manage them. Managed drives, what is currently being sold today, handle all the work of SMR themselves, transparently. Especially if you have a window into the drive and a high-speed camera, joked Christoph, referring to a paper that reverse-engineered a managed SMR drive [3]. There is another SMR type, host-aware, where the drive provides some information to the device driver, so you can optimize your drivers to work with it.

Someone thought SMR was an interesting idea but didn't like the interface the vendors were talking about for the host-aware. Christoph agreed, saying it was not a good fit. A vendor could build an in-kernel layer to support host-aware drives, if they were interested. So far, work on host-aware is actually merged into the kernel, but it is like the T10 (object storage standard) code: bit rotting because it is relatively unused.

Remzi Arpaci-Dusseau (UW) asked Christoph about other areas where he wished people were doing more research, and Christoph immediately replied practical cache management algorithms, backed up by James who asked for useful cache work. Christoph elaborated, saying that there has been very little research into multiple stacks of caches. Don summarized these points as practical cache management, algorithms for multiple interacting caches, and asked for more specific examples. Christoph responded that the inode cache is the "mother of all caches," then the kernel caches for data and pages, and the really interesting one, the directory entry (known as dentry) cache (also called dcache). Anything in the inode cache must have a corresponding dentry. When memory pressure forces cache evictions, inodes can't be freed unless there are no dentries referencing them.

Greg asked why dentries were so critical, as they only have to be used for system calls that use pathnames. Christoph replied that certain applications make a lot of use of pathnames, like build farms and unpacking zipped files. James added that most of the standard operational ways of handling files use names; that's why the Postmark benchmark is so important, as most mail servers manipulate many files by name.

Don followed up, asking about interactions with device level caching, and Christoph said they didn't have much data for that. Don observed that they have a lot of problems when memory pressure occurs with determining which cache entries can be thrown away, and throwing away the wrong entry is a problem. Greg added that it's not so much stacked caches as cache interactions that are the real problem.

Christoph named the kernel code used to reclaim memory as the shrinker. The shrinker has a direct hook into every cache, a routine in each cache handler that is supposed to create free space by releasing cache entries. And the space freed needs to be in large blocks that can be passed to the slab allocator. James explained that entangled knowledge, links between caches, caused a lot of problems for the shrinker. Someone wondered why not just use Least Recently Used (LRU) algorithms for shrinking caches. Greg explained that if you remove an inode entry, you also have to remove all the pages and dentries that the inode entry refers to. Christoph agreed, saying that the kernel does use LRU, but if a cache entry has lots of dependencies, it gets put

back into the queue of entries, while entries without dependencies get freed. Robert Johnson (SBU) restated this by saying that you can throw away the oldest stuff, but not the oldest stuff that has dependencies, a description that James agreed with.

Don pointed out that what you actually want back are pages. Ted Ts'o (Google) replied that certain objects are more likely to be pinned, so if you want page level LRU to work, directory inodes and dentries need to be on their own pages. Robert asked about page size, which is usually 4K, and James pointed out that Intel doesn't use powers of two for page size, which can either be 4K or 2 MB currently.

Christoph attempted to start the proposal discussions for the fourth time, but we took a short break instead. When we returned, Vasily Tarasov (IBM Research) suggested using a file system instead of caches, but Christoph said that this makes the problems even more complicated. All file systems use the core VFS (Virtual File System) for common services. Vasily pushed his point, but both James and Ted Ts'o said, "No." Ted Ts'o said that items that are referenced must be there and can pin a page. Christoph mentioned that compression might be used on inactive entries, but that would involve following pointers or using table lookups to find entries. The kernel developers had tried different structures, such as trees, but once there is a reference count, there are pointers between entries.

Vasily asked whether FUSE could be used to experiment with caches, but Ted Ts'o responded that FUSE uses caches heavily as a deliberate design choice, making FUSE a poor candidate for experimentation. Christoph suggested writing your own user space VFS, but it would be simpler to work with the kernel. Ted Ts'o then explained that it would actually be easier working within the kernel, as there are very rich debugging tools, much better than in user space. Try using gdb to debug a highly multithreaded program.

Jun He (UW) asked about conflicts that can cause thrashing. James replied that the kernel is a highly interactive system, and that kernel developers do their best to avoid thrashing. The best way to create thrashing is through memory pressure.

Someone pointed out that Linux is the only OS that uses a dentry cache. Christoph agreed that Linux has a strong dentry cache. Every open file has a filename associated with it, which helps developers with the problem of locking in cross-directory domains. Because of links, a file can have multiple parents, and you could deadlock when manipulating names. This is an area that can be extremely racy. James added that in UNIX, we think of files as names. In Linux we started out with inodes as the primary object. It would be a massive effort to change that. Ted Ts'o also weighed in, saying that this design allows a lot more of the complexity to be pushed up into the VFS layer, and all the details of locking can be down in the core VFS instead of in the underlying file system. Locking is one of the hardest things to do right.

Don asked about the difference between how Linux and BSD perform name lookups. While all systems start by passing the entire path to namei(), BSD and other System V-based UNIX systems process the path, while Linux iterates over each component of the path. Christoph answered that the VFS has to do work for each pathname component anyway. Ted Ts'o added that if the file system is left to figuring out the inode specified by a pathname, the file system has to understand mount points, which can cross file system boundaries.

The Proposals

While all of the previous discussion might appear to be a big distraction, a goal of the Linux developers is to help potential kernel developers become involved. There is an existing culture, as well as process, for making changes to the Linux kernel, and people who wish to make changes need to understand both.

I also asked Ric Wheeler (Red Hat), who started the workshop, but missed this one because he was snowed in near Boston, how he would describe the workshop:

"The purpose of the workshop is to get the Linux kernel community and the FAST file and storage research communities to know each other and our broad portfolio of work. On the FAST side, a large chunk of [research] is Linux kernel based and it is a challenge to know who to talk to in the Linux world and what the Linux community is up to. It is also good to know if the Linux community has already fixed a problem before a poor grad student launches a thesis built around a solved issue."

Proposal 1: Non-Blocking Writes to Files

Daniel Campello (FIU) presented the first proposal, based on his FAST '15 paper [4] that attempts to solve the blocking write problem. When a write occurs that will modify a portion of a block, the process has to sleep until the page is brought into memory. In their research, they store their changes someplace in memory until the page arrives, and then they patch the page before unlocking it. Daniel then asked what the kernel developers thought about this approach.

Christoph started by asking about real life workloads that cause these small writes. Daniel didn't answer this question. Ted Ts'o suggested that they measure with a whole bunch of benchmarks to see whether this is a huge win. Perhaps it would help with BitTorrent, which often receives blocks out-of-order. Daniel stated that they did put traces on writes, they checked for hits or misses, and most applications were weakly consistent. Every time an app wants to write, they recorded the size of the write. They used the SpecSFS benchmarks, and the results seemed to match very well: 30% of writes were not aligned to page boundaries.

Christoph asked how unaligned are the writes, but Ted Ts'o took a different approach: the application is misbehaving. Ted used the example of the BFD Linker, which does unaligned writes, as opposed to the Gold Linker, which doesn't. Use the Gold Linker, said Ted, or fix the app. If they add complexity into the kernel,

they have to support it forever. It is easier to fix the few applications that make the kernel more complex for corner cases. Christoph suggested that another idea would be to simplify things: get rid of the struct buffer head, one buffer head per page, and you could have multiple buffer heads per page.

Christoph was a bit more positive overall than Ted, saying that they could use this research to track changes at the byte level to support these types of apps. Daniel said they use a patching mechanism, with the changes hanging from the page headers.

Raju raised another question, about something they noticed during their research, and asked what type of information the developer would like to have about it. Ted suggested tracepoints, which give you the process and username, and said it's not that hard to add a tracepoint to the kernel. Christoph offered to write the tracepoint for them. Raju asked whether this is systemtap, but Ted replied that this is different, and that some subsystems are already heavily tracepointed. The tracepointing system is very efficient by design.

Daniel continued by saying that they collected this data on the file server for both Linux and Windows systems, including a Linux Web server. Daniel thought that their non-blocking writes really solved an important problem. But Ted was not so sure and wanted to understand what was going on at multiple levels of the storage stack to cause this. He worried that there is something important that the developers were not understanding. Daniel mentioned that they ignored files that have holes, and wondered how often files contain holes (regions not yet written to). Christoph answered that it is common for HPC and for VM images but is rare on desktop systems.

Ted continued to worry that they might be missing something important that causes this behavior. Daniel offered to get more information and said that they had avoided collecting too much because of privacy concerns. Raju wondered whether other people had traces that could help. Ted suggested that SQLite could be doing unaligned writes, or MS Word over SMB could be doing it, as well as BitTorrent. Daniel mused that perhaps one of these cases involves a process writing a small file and then going back and overwriting the same page. Greg suggested it happens with large files, when appending without reading first.

Ted said that many kernel developers don't take file servers seriously: cloud servers and personal desktops, yes, but not file servers. There are a set of commercial companies who care about this: those doing small NAS servers. But the kernel developers care more about servers that scale out. Greg imagined a variety of things that could do this, like a logfile that rarely gets written, and its page gets evicted. Daniel said that the Chrome browser actually does this a lot, writes a small file, then overwrites it, every ten minutes. Ted argued that that's a buggy application: Chrome should truncate then overwrite, or write a new file then do an atomic update. The app is doing something that is inherently unsafe.

Erez had a quick follow-up. Key-value stores need to be really efficient, use aligned pages, and use workloads designed to overwrite an entire page, but the kernel still brought in the entire page. James replied that Erez wants trim for the page cache, before Christoph suggested they move on to the next proposal, BetrFS [5].

Proposal 2: BetrFS

Don said that the code they have right now is not ready for prime time, and they wanted to get a sense of what should be done next. Christoph got right to the point: Linux already has over 100 file systems, so you must get people excited about your new file system. Don responded that they like B-epsilon trees, and they plan on writing an article about B-epsilon trees [for *;login:*]. B-epsilon trees are generally an order of magnitude or more faster for inserts than B-trees.

Ted pointed out that kernel developers are very practical. B-epsilon trees are very cool, but what's not immediately obvious is the killer app that needs them. For example, flash file-systems developers have huge commercial imperatives to stabilize those file systems, using 50–200 person years. But for BetrFS, he's not sure why anyone would invest that amount of time. Robert replied that fractal trees [another name for B-epsilon trees] were invented as a backend for MySQL. Using BetrFS with MySQL might also result in 10–30 times the performance.

Christoph suggested that they try to fit their project into an existing file system, then kernel developers could see the advantages by the one-to-one comparison. Ted wondered whether B-epsilon trees would fit well into the BtrFS, but he didn't know about the impedance matching. But the BtrFS folks would like a faster B-tree.

Robert asked whether BtrFS uses a key-value store internally, and Christoph answered that B-trees are used in some places. Robert emphasized that B-epsilon does much faster inserts, and suspected that it will fit into the kernel somewhere. James said that usually they have a problem kernel developers need to solve, and the Stony Brook researchers have a technology looking for a problem. Robert said that they are certain there are problems, and that this technology changes the performance landscape.

Where in file-system land would it be doing massively more updates than queries, wondered Christoph. Robert suggested that every read results in a write because reads mean updating the atime. Someone pointed out that that is why atime updates are disabled by default in Linux. Another person suggested that SQLite does lots of small writes, and Christoph replied that SQLite should have a sensible reimplementation.

Ted asked again about places in the kernel where inserts are more common than queries; he believes this is more common in apps. Erez suggested archival and backups, which Christoph dismissed as write-once-read-never. Greg said that you must read before you can write with backups, while Robert kept pushing, saying there must be a key-value store that will benefit from B-epsilon trees.

Proposal 3: Dcache Lookup

Don said that they had been looking at dcache for a long time. Dcache is used to speed up pathname-to-inode lookups by caching info from past lookups. Don had wondered whether they could speed the process up, and found they could indeed, achieving a 5–26% performance increase.

Don proposed to speed up lookups of non-existent files (which happen, for example, whenever a new file is successfully created) by adding a flag to dcache entries for directories. The flag would indicate that "the dcache contains a complete list of all the entries in this directory." This flag would enable a lookup that fails to find an entry in the dcache to return a definitive "NO." Don described how to initialize this flag as part of a readdir() call and pointed out that whenever an application created a new directory, the flag could be set to TRUE. This could dramatically speed up processes that create an entire directory hierarchy, such as git checkout or untar. Currently, without this optimization, the kernel queries the underlying FS for each file and directory created, even though the parent directory was just created a moment earlier and the kernel has a full cache of its contents. Then Don asked whether the developers would be open to trying new lookup algorithms.

Christoph said simply: send your code to Linus now. Ted said that Linus is really interested in improving path lookup and complains when file stats are slow. Don said that they measure a 26% improvement in getting file status. Christoph repeated, send it to Linus, although Linus will rewrite whatever you send him. Don said he doesn't care, and pointed out that they were compatible with SELinux. Christoph asked whether they still have slowpath, and Don said they do, but worried about their patch being slower when there is a rename in an early directory in a long pathname. Greg said that renames near the beginning of a path don't happen often enough to optimize for, and Ted suggested including a version with the cached info, so it could be invalidated if needed. Christoph felt some concern about getting the readdir() case right, but that the mkdir() case seemed simple enough and gave a path for incremental deployment.

Robert presented the second part of their proposal, to speed up readdir(). They want to keep negative results and add a bloom filter. The bloom filter would be kept when memory pressure results in freeing cached dentries. Ted worried about the memory used, and Christoph said this could be a good research paper.

Proposal 4: Chopper

Ted asked for a recap from the lead author of Chopper [6]. Jun He (UW) said that they had tried their favorite workload and found some problems in the block allocator. They wanted to search the design space for file systems and needed a tool to help with experimental design. They developed Chopper as that tool and uncovered four different issues with ext4.

The first issue involved scheduler dependencies in ext4, when many small files were created in a single directory. Ted asked about the real world scenario where this happens; perhaps if the app tried slow writes using multiple CPUs.

Jun plunged on with the details of their second finding, when four threads are writing to one file. Again, Ted interrupted Jun, saying that they must have delayed allocation disabled, and that examples must be real world apps, not synthetic tests. Ted went on to say that he found their testing framework, Chopper, interesting and that perhaps there were other metrics he'd like to test, such as the average size of free extents. He worried not just about the goodness of written files, but also the goodness of the free list. Another issue would be a change that helps metric A but causes regression in metric B.

Jun never got through all four of their findings. Instead, we broke for coffee, with just 90 minutes remaining in the workshop.

Proposal 5: A New Form of Storage Virtualization

Zev Weiss (UW) described the work in their paper [7], which allows adding features from more advanced file systems, like ZFS and BtrFS, to ext4 and XFS. The idea was to work at the block layer to add features (e.g., snapshots, copy-on-write) transparently to existing file systems. Christoph said that if the researchers could take this further, it could be helpful and very interesting. Ted said that he had written up a proposal for a device mapper for a block, and that there were interesting things that could be done at the device mapper layer. You do need to have some communication between the device mapper and the block layer, but this would be interesting to explore.

Don asked about the standard for getting a device mapper into the kernel; Christoph said this can be done, and Vasily has done this.

Proposal 6: Deduper

Vasily Tarasov (IBM Research) presented his idea for an open source, block-level deduper, at the 2014 Linux FAST workshop. Vasily worked with Christoph, who helped him with his patch set and with submitting the patch set to the device mapper list. They immediately got back the response that they needed more comments in their patch set. In July 2014, Vasily presented a paper [8] on a device mapper deduper. In August 2014, they submitted another patch set, and got some attention, but not a lot of responses. In January 2015, one developer became devoted to this module, working to fix things, and in that respect the process has been good. The amount of work required to keep the process of getting the patch into upstream has varied from month to month.

Christoph asked Erez about the Stony Brook students' poster about dedupe. Erez said that not all data blocks are created equally, that some are metadata. If you are submitting metadata, you want to set a flag for "don't dedupe." A metadata hint was what they needed, and they found that such a flag was there. Christoph said that hint was there for tracing, and could be very useful. Ted asked which file system has Sonam Mandal (not present) worked on? Erez replied that she had worked on all the

exts. Ted said that he thought ext4 was well marked, and Erez replied that they are also looking into pushing hints further up the storage stack for temp files and for files that will be encrypted.

Ted said they are interested in discovering which hints have the highest value so are worth implementing. You want to set these hints in the inode, not in the page cache. Christoph worried about trusting users with these hints, as they could be abused or simply used poorly. Erez wanted to put the ability into libc, and Ted thought that such hints would be useful for compiler temp files.

Erez asked about the existence of a metadata flag, and Christoph said it was only used for tracing. That flag used to be in the I/O scheduler, and is still present in ext2 and ext4. Ted said the flag is there for the commit block, and nothing can go out until that commit block has been committed. Erez joked that they didn't want to dedupe the duplicate superblocks, and people laughed. They treated these duplicates as metadata, and wondered whether they should submit patches that do that. Christoph replied, "Sure."

Ted continued the discussion by focusing on priority inheritance in a scenario involving a low priority cgroup process wanting to read a directory, locking it, but being unable to read it, and then a cluster manager needs that block but can't read it. Raju said they found this problem in their work on non-blocking writes, but didn't fix this behavior. Ted said that read-ahead has a low priority, and if you have a process blocked because the process needs that block, that really causes a problem. The right answer is that we need to have better visibility. And the real problem is that these problems have been hiding in the underbrush for years.

Raju asked whether this was a significant enough problem for people to research. Ted replied that if this is a problem on Android, but fixing it would break big servers with high performance I/O, that would cause problems. They won't risk server performance to make Android faster. Samsung can apply that patch locally [9] for their handsets, but we can't push it upstream.

Zev asked what type of interface was imagined for things like block remapping. Christoph replied that they have thought about operations for block devices for a long time. Zev then asked about stable pages, where the page is locked. Ted said they just make a copy of the block as a hack, and that they just worked around it, the copy worked acceptably, but the patch cannot be pushed upstream. James said he has seen a lot of email complaining about the problem. Ted replied that they could send out the patch to see whether there is any interest.

Discussion

We had finished with the formal proposals by 5:15, and Christoph opened the floor to general discussions.

Dungyun Shin (KU) wants to measure times in the local I/O stack performance so he can then learn which layer is causing a problem. If he knows where the problem is, he knows what needs to be optimized. Dungyun asked for feedback on which layer to focus on. Dungyun is currently working on the block I/O layer and the VFS layer.

Christoph said there are two parts to this question. You can get perfect info using blocktrace, and there is a tool there for measuring latency. There are also tools at the system call level, but the problem is combining the two.

Ted said there has been recent related work at Google. They were interested in long tail latency—2, 3, 4 nines latency—and they measured worst-case times, and didn't try cross-correlating. They just wanted to find where the long tail was happening. For them, CFQ (Completely Fair Queueing) was completely out to lunch, or in the hardware, the drive was going out to lunch for hundreds of milliseconds. Rather than trying to cross-correlate all the way down, it was easiest to measure the long tail.

Dungyun asked whether it is a good strategy to correlate interrupt times, since some benchmarks running on Android are very variable, not consistent. They can't trust the research because of this. Dungyun tried experiments to eliminate the time spent I/O handling, and then he got more consistent results.

Ted said he doesn't have a high opinion of many of those benchmark scores—do they actually reflect what the user will see? Measuring how long it takes to start the Facebook app, let's measure that because that app is really big. Handset vendors actually check the application ID so the CPU benchmarking scores can be artificially high. Those scores are there for "benchmarketing." Facebook reads 100 MBs before it displays a pixel. If you want to find out what is actually causing the delay—memory pressure, locking—measure latencies at different levels of the I/O stack. But then, why are you optimizing this?

Don asked about the developers' perspective and the application developers' perspective. Ted countered by asking whether this is the application you really care about? What is the real world case where we are not optimized for this area? If he decided to fix this in ext4, but people who run into this are using xfs, why should he fix this? When thinking about writing papers, pick something that will be high impact—that is, something that will affect lots of users. That's why Ted liked the Quasi I/O paper [9]. But does three seconds instead of one second really matter to the handset user?

The meeting broke up just before 6 p.m., because Google needed to secure the room they had loaned us.

References

[1] Info for Linux developers mailing list: http://vger.kernel.org/vger-lists.html.

[2] Linux Storage Stack Diagram, Werner Fischer: https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram.

[3] A. Aghayev and P. Desnoyers, "Skylight—A Window on Shingled Disk Operation," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/aghayev.

[4] D. Campello, H. Lopez, L. Useche, R. Koller, R. Rangaswami, "Non-Blocking Writes to Files," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/campello.

[5] W. Jannen, J. Yuan, Y. Zhan, A. Akshintala, J. Esmet, Y. Jiao, A. Mittal, P. Pandey, P. Reddy, L. Walsh, M. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. E. Porter, "BetrFS: A Right-Optimized Write-Optimized File System," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jannen.

[6] J. He, D. Nguyen, A. Arpaci-Dusseau, and R. Arpaci-Dusseau, "Reducing File System Tail Latencies with Chopper," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/he.

[7] Z. Weiss, S. Subramanian, S. Sundararaman, N. Talagala, A. Arpaci-Dusseau, and R. Arpaci-Dusseau; "ANViL: Advanced Virtualization for Modern Non-Volatile Memory Devices," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/weiss.

[8] V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, and E. Zadok, "Dmdedup: Device Mapper Target for Data Deduplication," OLS 2014: https://www.kernel.org/doc/ols/2014/ols2014-tarasov.pdf.

[9] D. Jeong, Y. Lee, and J. Kim, "Boosting Quasi-Asynchronous I/O for Better Responsiveness in Mobile Devices," FAST '15: https://www.usenix.org/conference/fast15/technical-sessions/presentation/jeong.

REAL SOLUTIONS FOR REAL NETWORKS

FREE CD or DVD in Every Issue!

Each issue delivers technical solutions to the real-world problems you face every day.

Learn the latest techniques for better:

- network security
- system management
- troubleshooting
- performance tuning
- virtualization
- cloud computing

on Windows, Linux, Solaris, and popular varieties of Unix.

6 issues per year!

ORDER ONLINE AT: shop.linuxnewmedia.com