ZFS Fast Dedup

Allan Jude
Co-Founder
Klara Inc.
Ontario, Canada
allan@klarasystems.com

***Abstract*—The deduplication feature was added to ZFS version 21 and later incorporated into FreeBSD in early 2011 then first released as part of FreeBSD 8.3. In the more than a decade since then, users have been advised to avoid the feature due to very poor performance. We present a series of enhancements to ZFS deduplication to improve performance and make the feature useful for general purpose storage and for high density VM hosting.**

Keywords—ZFS, deduplication, dedup, filesystem, storage, FreeBSD, performance

I. Introduction

Demand for data storage continues to grow at a more and more rapid pace. While cost per TB stored continues to decrease, any technology that can further reduce the amount of storage consumed by the various workloads on a system has a wide and eager audience. ZFS features transparent compression which can offer savings on certain types of data, and deduplication which promises to avoid storing the same data multiple times, possibly resulting in massive savings. However, the promise of the dedup feature is diminished by the performance tradeoff, which can be quite extreme, as the size of the deduplication table grows, write performance continues to decrease until the storage is nearly unusably slow.

II. What makes legacy dedup slow

In order to implement deduplication, ZFS must keep track of the strong cryptographic hash of every block that has been stored. When a new block is written, its hash is calculated as part of the normal ZFS write process, but before the block is stored to disk, the list of all existing blocks is consulted to see if an identical block is already on disk. If it is, rather than storing the new block on disk, its metadata instead references the existing block, and the entry in the list of blocks is annotated with an additional "reference" to that data. When a block is deleted, the list of blocks must be consulted, and one of these references is removed, only once the number of references reaches zero can the disk space be reclaimed.

Enabling deduplication has a number of effects on how ZFS stores data. The first of which is changing the checksum algorithm from the default fast fletcher4 checksum, which is not judged to be collision resistant enough, to sha256. This increases CPU usage but is generally not a large contributor to the loss of performance unless the number of available CPUs is low. The biggest impact comes from the fact that in order to write new data to disk, ZFS now needs to consult the list of hashes of existing blocks to determine if the new block actually needs to be written or can reference an existing copy. The list of hashes is stored sorted by the hash, to allow faster lookups.

To implement the list of hashes of existing blocks, ZFS stores two DDTs (deduplication tables) on disk. These are large key-value databases called ZAPs (ZFS Attribute Processor), which are sorted by the key and allow for relatively fast lookups. Each entry's key consists of the checksum of the physical data, its logical size, physical size (after compression), compression algorithm, and some other properties. The value is an array of four entries describing where the data is stored on disk, how many unique blocks reference it, and the time when it was first written. ZFS keeps one DDT for all blocks that are unique (have never deduplicated), and one for blocks that are duplicated.

When new data is written, ZFS will prepare the data to be written to disk as usual, including compressing it, and calculating a checksum. Before allocating space on disk, ZFS will lookup the calculated checksum in the DDTs to see if a copy has already been written to disk. If a copy does exist, the existing entry is updated to increment its reference count, and the entry may need to move from the "UNIQUE" ZAP to the "DUPLICATE" ZAP. If the checksum is not found in the DDT, then it is inserted into the "UNIQUE" ZAP.

Due to the nature of strong checksum algorithms (SHA256 is used by default for deduplication), when a number of blocks are written sequentially, each will have a hash that does not resemble the others. Since the ZAP is stored in sorted by the key to improve search speed, this means the lookups of these new pending writes must access very different sections of the DDT. This means every new write results in 1 to 4 random reads from the DDT (to load any indirect blocks to find the location of the desired entry within the ZAP) to determine if the data already exists.

Once the deduplication status of the newly written blocks has been determined, if the data is unique, it must be written to disk, and then, whether it is unique or not, the DDT must be updated to either insert the new entry or increment the reference count of the existing entry. This results in a further 1 – 4 writes to disk to update the ZAP and its indirect blocks. The DDT is a critical data structure in ZFS, so all of its blocks are stored with copies=3, increasing the inflation further.

In a pathological case, a database application writing 100 operations per second, could result in as many as 400 read IOPS and 1300 write IOPS, a whopping 1700% workload inflation.

III. Fast Dedup

We introduce a different way to handle updates to the DDT ZAPs, to reduce this inflation. Rather than updating the DDT at the end of each transaction group, we instead create an in-memory AVL tree and corresponding append-only on-disk Fast Dedup Table log (FDT-Log). When a new or updated DDT entry is created, rather than writing the change directly to the ZAP, the updated entry is inserted or updated in the AVL tree and appended to the on-disk log. When the AVL tree reaches half of the configured maximum memory usage, it switches to the flushing state, and a new empty AVL tree is created in its place. The AVL trees in ZFS have been modified to be sorted in the same order as the DDT ZAP, as previously they were sorted by the checksum but in 16 bit chunks

resulting in different ordering. With the AVL tree sort matching the on-disk DDT sort, ZFS is able to write out larger batches of changes with better aggregation and amortizes the cost of the updates to the indirect blocks. This batching reduces the total amount of metadata that needs to be written. Over the next few transaction groups (rate controlled to avoid overloading the system), all of the pending changes in the AVL tree are written to the DDT ZAPs, and then the flushing AVL tree is freed.

The on-disk FDT-Log works much the same way, except due to its append-only nature, the log is not sorted. When the in-memory AVL tree is full, a new log corresponding to the new empty AVL tree is created, and any future changes are written to that log instead. As the full AVL tree is flushed to the DDT, each transaction group the bonus buffer of the old FDT-Log is updated to reflect the checkpoint, the last hash that was successfully updated in the DDT. When the old AVL tree is empty, the old log is freed.

In the event the system is shutdown or crashes, the in-memory AVL tree can be recreated from the on-disk log, replaying the log entries sequentially to perform the inserts and updates to the AVL tree, and then using the checkpoint in the bonus buffer to resume flushing the AVL tree to the DDT where it previously left off, preventing old updates from being repeated.

IV. Further reducing inflation

One of the biggest contributors to the inflation caused by deduplication is the fact that the DDTs are stored in triplicate because they were deemed critical to the operation of the pool. When a block is deleted, if its block pointer's deduplication bit it set, ZFS updates the DDT to reduce its reference count, and if it is the last reference, allows the space to be reclaimed. Before fast dedup, ZFS would panic if a block with the dedup bit set could not be found in the DDTs.

After discussing the issue at length with the ZFS Leadership Team, we concluded it is safe to reduce the DDTs to copies=1 and apply the same policy as normal data for the indirect blocks, that is, keeping additional copies only of the higher level indirect blocks that impact a large number of blocks. It is assumed that those operating deduplication will provide redundancy at the pool level with mirrors or RAID-Z rather than relying on the best-effort copies mechanism.

Additionally, the invariant that any block with the dedup bit set must be able to be looked up in the DDT was removed as part of the FDT Pruning work described below. If a block has the dedup bit set, but is NOT present in the DDT, it is assumed that it was a unique block with only a single reference, and so is safe to reclaim.

Additionally, we have added logic to handle the possible corruption of the DDT as elegantly as is possible. If a hash cannot be looked up in the DDT due to a read failure, the space is never freed to avoid destroying the data that may be in use by other referents. This would effectively "leak" the space, meaning it can never be freed, but the damage would be limited to the single DDT block, which could at most impact fewer than 500 blocks.

V. Other Enhancements

In order to make dedup useful, Klara created a number of additional features that give the operator more control over the performance tradeoffs of deduplication.

A. Dedup preload

The cost of the random reads to determine if a block has already been written can be very high. Keeping the DDT cached in ZFS's Adaptive Replacement Cache (ARC) can almost entirely mitigate this cost. However, after a reboot the cache will be cold, and performance will suffer.

One production customer of ZFS had a DDT of over 100 GiB, which meant after a reboot the system performed poorly for at least 72 hours, until most of the DDT have been pulled into the cache by the random reads generated while writing newer data.

The new dedup preload feature allows the operator to explicitly warms the cache by sequentially prefetching the entire DDT. Optionally, if the ARC is not large enough to support the entire DDT, this feature can prefetch only the indirect blocks, to keep the read inflation to a maximum of 100% (instead of 400%).

With these changes, system performance can be restored after a reboot in minutes instead of days.

B. ZAP Shrinking

The ZFS Attribute Processor (ZAP), the key-value pair system used internally by ZFS, does not support shrinking. If a directory, DDT, properties list, or any ZAP in ZFS grows to a large size, and then the number of entries is reduced, the corresponding size of the ZAP does not diminish. The ZAP will be sparse, and compression may reduce its size, but the number of blocks will remain at the high water mark, creating longer search and update times.

In order to implement the additional features described below, Klara implemented an enhanced version of ZAP shrinking to be able to reclaim space in the ZAP.

As a side effect of this work, Klara completed the in-progress work on ZAP shrinking that will improve the performance of formerly large directories and other ZAPs that have shrunk.

C. Dedup Quota

In order to limit the size of the DDT to ensure it fits in memory, or within a vdev of fast devices (NVMe), Klara added new functionality to allow the administrator to configure the maximum size of the DDT, so that it does not exceed the target size. If this maximum is reached, no new unique blocks are added to the DDT. Any data that is duplicate of existing data is still able to deduplicate.

When the legacy DDT was able to fit in RAM, or at least on fast media, performance could be acceptable for some workloads, but the moment the DDT exceeded that size, and some writes requires reading data from slow spinning media, the performance of the entire system was crippled. Giving the administrator the required controls to prevent performance dropping off a cliff as the DDT grows makes deduplication infinitely more useful.

Deduplication could also cause serious problems if the size of the DDT exceeded available memory during pool import and recovery. If a rewind or other operation requires writing to the pool, and the DDT was constantly being pages in and out due to being too large the system could take excessive amounts of time to bring the pool online, sometimes on the order of weeks due to the extremely poor performance caused by the workload being predominantly random reads. While

performance is greatly improved by FDT, the quota feature helps ensure the system does not exceed its own capacity.

D. FDT Pruning

In order to have the dedup quota feature be more useful, Klara also implemented FDT pruning. When the DDT is nearing its quota, ZFS will remove some of the oldest entries from the unique DDT. It is assumed that the oldest entries have the least chance of deduplicating, as they have been around the longest and have yet to find a match. Removing these older entries in favour of newly written data that may have a higher chance of deduplicating against yet newer data is judged to be more desirable.

This ensures that the deduplication functionality can continue to operate, and that the data most likely to benefit from deduplication being resident in the "unique" DDT.

VI. Performance

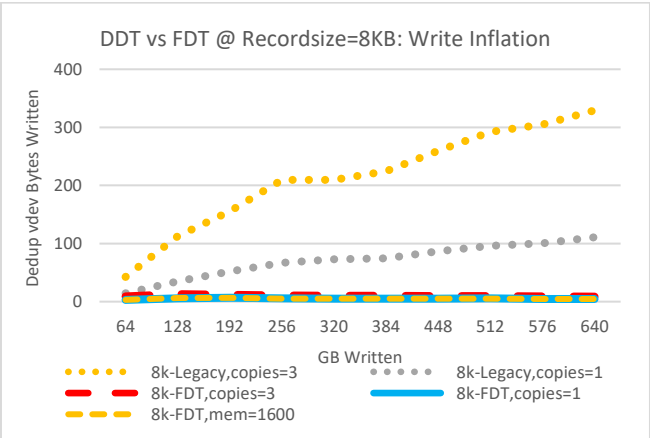
Legacy deduplication suffers from extremely poor performance, which gets worse as the size of the DDT increases. In the field, we have seen customers with DDTs containing 400 million blocks. Even with NVMe based storage, these pools perform extremely poorly for writes, due to the demand reads required for each write. The customer persists using dedup in spite of this, because they are storing 50 TiB of data (compressed to 37 TiB) using only 7.7 TiB of storage space with the savings from deduplication, even though 240 million of the 400 million blocks are unique.

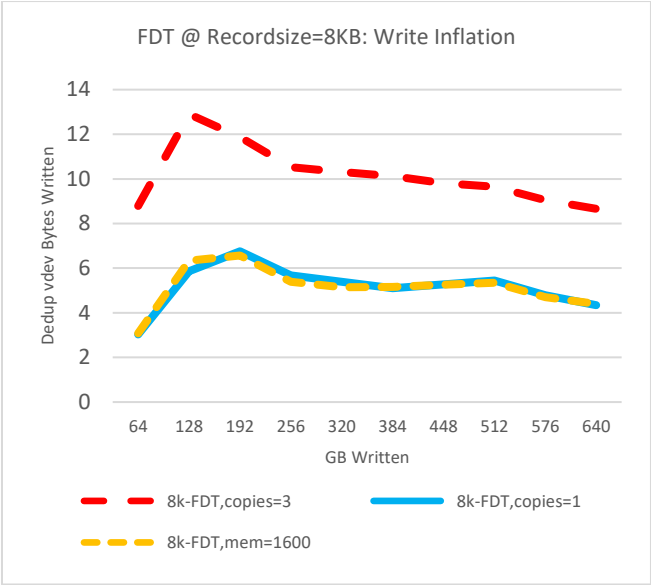
In the following benchmarks, we write data to a pool with 8 KiB records (to create a larger number of DDT entries), in batches of 64 GiB while measuring write throughout. After each batch, we drain the FDT-Log and then measure the total amount of data written to the dedicated dedup vdev, to measure the inflation factor. Batches are repeated to measure the change in throughput and inflation as the size of the DDT grows.

TABLE I. Reduced Amplification

Write Amplification		
Configuration	***Total Inflation***	***Incremental Inflation***
Legacy	311%	479%
FDT copies=3 maxmem=320M	15%	19%
FDT copies=1 maxmem=320M	8%	10%
FDT copies=1 maxmem=1.6G	7%	9%

FDT reduces inflation by 5x – 30x



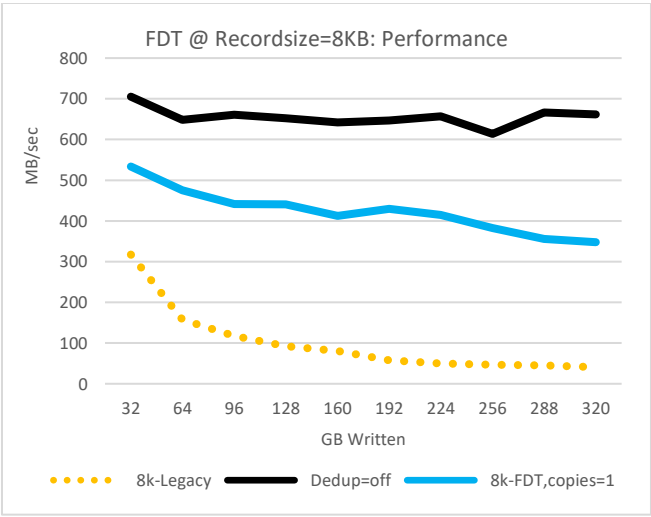


The most important improvement from FDT is the ability to sustain much higher write throughputs than with legacy dedup. As our results show, performance is maintained at a level much closer to a system without deduplication and does not suffer the same linear decline as the size of the DDT grows.

TABLE II. Increased Performance

Write Performance		
Configuration	***MiB/s***	***% of Baseline***
Legacy	44	100%
FDT copies=3	357	811%
FDT copies=1	362	822%
FDT copies=1 bs=32k	361	820%
No Dedup	647	1470%

FDT increases performance over legacy dedup by 800%



In these tests, all writes were unique blocks. With some portion of writes being obviated by deduplication, real-world throughput of FDT would be much closer to the speeds observed without deduplication.

VII. Conclusions

With the changes we have created, ZFS deduplication becomes a viable option for use in production without the extreme performance tradeoffs requires in the past. With new lower latency storage media, dedicated dedup vdevs, and the new suite of options and tunables, dedup becomes generally useful for a wide range of workloads. Fast Dedup improves performance of a previously pathological workload by over 800%

Acknowledgment

The author would like to express his thanks to the entire team at Klara Inc. who helped create and implement this new design and make it successful, especially: Rob Norris, Don Brady, Alex Stetsenko, Mateusz Piotrowski, and Fred Weigel.

The author would also like to express gratitude to the entire OpenZFS community who have been a pleasure to cooperate with and have been accepting of Klara's role as a steward of OpenZFS. Special thanks to those who helped refine the design of fast dedup: Matt Ahrens, Pawel Dawidek, Alexander Motin, and Rich Ercolani.

Lastly, this project would not have been possible without the financial support of iXsystems and the other sponsors of the Fast Dedup project.