Zstandard Compression in OpenZFS

BY ALLAN JUDE

ZFS is a highly advanced filesystem with integrated volume manager that was added to FreeBSD in 2007 and has since become a major part of the operating system. ZFS includes a transparent and adjustable compression feature that can seamlessly compress data before storing it and decompress it before returning it for the application's use. Because the compression is managed by ZFS, applications need not be aware of it. Filesystem compression not only saves space, but in many circumstances, can even lower read and write latency by reducing the total volume of data that needs to be stored or retrieved.

Originally ZFS supported a small number of compression algorithms: LZJB (an improved Lempel–Ziv variant created by Jeff Bonwick, one of the co-creators of ZFS, it is moderately fast but only offers low compression ratios), ZLE (Zero Length Encoding, which only compresses runs of zeros), and the nine levels of gzip (the familiar slow, but moderately high-compression algorithm). Users could thus choose between no compression, fast but modest compression, or slow but higher compression. Unsurprisingly, these same users often went to great lengths to separate out data that should be compressed from data that was already compressed in order to avoid ZFS trying to re-compress it and wasting time to no benefit. For various historical reasons, compression still defaults to "off" in newly created ZFS storage pools.

In 2015, LZ4 replaced LZJB as the default when users enable compression without specifying an algorithm.

In 2013, ZFS added a new compression algorithm, LZ4, which offered both higher speed and better compression ratios than LZJB. In 2015, it replaced LZJB as the default when users enable compression without specifying an algorithm. With this new high-speed compressor, combined with an existing feature called "early abort," it became feasible to simply turn on compression globally, since incompressible data would be detected and skipped quickly enough to avoid impacting performance. The early abort feature works by limiting the size of the output buffer given to the compression algorithm to one-eighth smaller than the input buffer size. If the compression algorithm cannot fit the output into that smaller buffer, it fails and returns an error. As a result, the algorithm can preemptively disengage if it is not going to provide sufficient gains, in which case the data is stored uncompressed to avoid the overhead of decompressing a block that was barely compressed. In fact, enabling LZ4 compression on everything is so low impact that this set-and-forget configuration is very common and has even been the default in FreeNAS for many years.

The Project Begins

The project started in the fall of 2016 after the author had to miss the OpenZFS Developer Summit due to a scheduling conflict with EuroBSDCon. The goal was to integrate a recently announced new compression algorithm into OpenZFS. Zstandard (Zstd for short) was created by Yann Collet, the original author of LZ4. The purpose of the new algorithm was to provide compression ratios similar to gzip (with even greater flexibility, offering more than twenty levels to gzip's nine!) but with speeds comparable to those seen with LZ4.

As the project began, we immediately ran into issues with stack size since Zstd was written as a userspace program and had a penchant for large stack variables. This was a problem for integrating Zstd into the kernel, where the stack was limited to 16 KB, and had to support all of the other layers of the operating system before and after the compression integrated into the filesystem. We temporarily sidestepped this problem by just increasing the stack size in our development kernel and got the first version of ZFS with Zstd compression working after a few weeks of work. Then we set about modifying Zstd to instead use heap memory returned by the kernel malloc framework to reduce stack usage. This was difficult as there were often multiple exit paths from functions where the allocated memory needed to be freed. After only limited success, the project was set aside for a while, knowing when we came back to it, it was likely to be even worse, as all of the local patches would need to be rebased forward to a newer version of Zstd.

Luckily, when it came time to return to the project, Zstd version 1.3 had been released, with greatly reduced stack usage, while also allowing the caller to manage their own memory allocation. With these welcome improvements, Zstd would no longer require extensive modifications for kernel integration. In the end, only superficial changes were required, and Zstd could be used largely unmodified.

By the fall of 2017 and the next OpenZFS Developers Summit, we had a working prototype to demo at the conference. The summit provided an invaluable opportunity to talk to experts and much more experienced developers about remaining challenges. One of these was how to have the user provide the compression type (Zstd) and the level (1-19) in a way that would not result in fatal confusion should a user later change the compression type to gzip, where a level like "19" might be invalid. This issue was mentioned during the talk, and afterwards Robert Mustacchi came up and suggested a remarkably elegant solution: only expose the compression type to the user offering the different levels of Zstd but store them internally in ZFS as separate values. While that whole conversation took less than two minutes of his time, it saved many weeks of work. During the breaks, we also talked to a few people about any ideas they might have to solve other issues, and what uses they might have for Zstd.

We presented our progress at BSDCan 2018 and there was a good deal of interest. Though there was still much to be done before it could be committed, the prototype showed how much benefit Zstd could provide to ZFS and FreeBSD.

Beyond the Prototype

After getting the initial functionality working, there were larger integration issues to address. How will this all integrate into ZFS? In the ZFS on-disk format, the compression type is stored in an 8-bit field in each block pointer. The top bit had already been borrowed to represent embedded block pointers, for the case where a block compresses so well (112 bytes), that it can be stored directly in the block pointer in place of the disk addresses and checksum, and therefore does not require its own allocation on disk. This means that no more than 127 compres-

sion algorithms are possible, and another bit may need to be borrowed in the same way in the future. A number of slots are already used: The value 0 does not actually mean no compression, it indicates that compression is inherited from the parent object. With levels for on, off, lzjb, empty (a whole block consisting entirely of zeros), gzip 1 through 9, ZLE, and LZ4, the first 15 values are already used. In the end, this Zstd patch introduced 41 additional compression levels (1-19, "fast" 1-9, "fast" 10-100 in increments of 10, "fast-500" and "fast-1000"), which could lead to very few possibilities left in the compression field in the on-disk format. After examining how the compression field in the block pointer is used, it became clear that the on-disk format only needs to map the compression setting to the correct decompression function, which is the same for all Zstd levels. At the time, it did not seem like it would be necessary to store the specific level of Zstd a block was compressed with.

After further work, it was discovered that sometimes we actually do need to know what level a block was compressed with. Namely, in the (presumably infrequent) case where the compressed ARC feature is disabled, the L2ARC would consistently fail with checksum errors. The L2ARC is a second-level cache that copies data at risk of being evicted from the primary ARC. By design, the L2ARC avoids the overhead of keeping its own copy of the checksum of each block, and instead refers to the checksum in the original block pointer. This means each block must be recompressed with the exact same settings before being written into the L2ARC. When reading back from the L2ARC, the block is checksummed and compared to the on-disk block and the original checksum. With the previous compression algorithms, there were no additional parameters to consider, but with Zstd, recompression at the default level would most likely generate a different output, and therefore mismatched checksums.

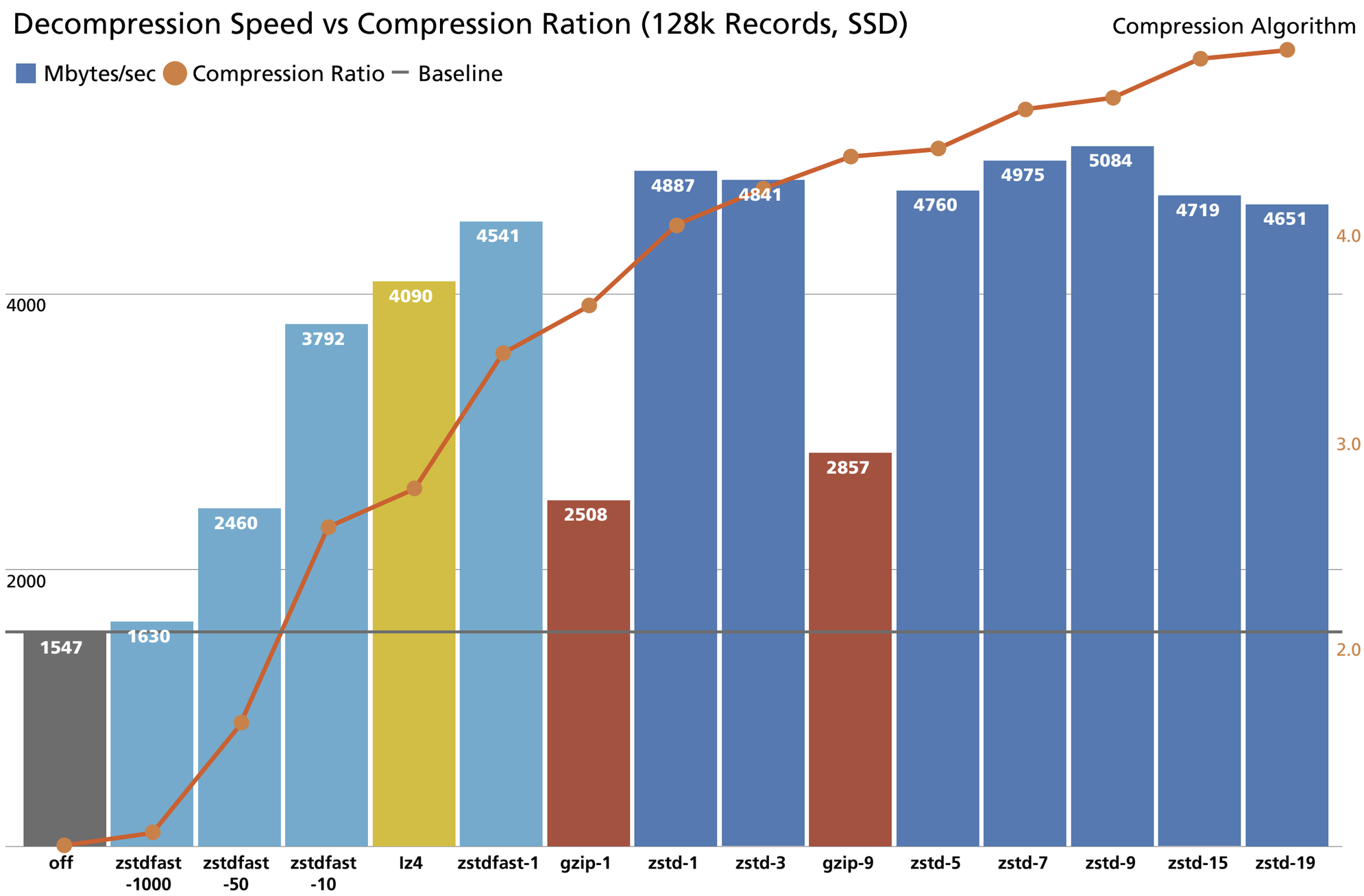
After further work, it was discovered that sometimes we actually do need to know what level a block was compressed with.

To solve this, we extended an existing concept used in LZ4, where the first 4 bytes of a compressed block on disk are used to store the compressed length of the block. Since allocations on disk will always be whole sectors, this allows LZ4 to avoid reading and attempting to decompress the random data in the slack space between the end of the compressed data and the end of the sector. Zstd compressed blocks use a larger header and store the version of Zstd and the level of compression in addition to the size. We decided to store the version of Zstd used to make it easier to upgrade the version of Zstd in the future, giving us the possibility to include multiple versions of the Zstd compression functions, so that a block could always be recreated if required. This is most likely to come in handy for the "NOP-write" feature: when a block is to be overwritten, ZFS can compare the checksum of the new block, and if it is the same as the old block, it does not need to rewrite the data. This type of operation is very common with Oracle databases and may also happen with certain types of backup software. If the original block is compressed with an older version of Zstd but now recompressed with a newer version, this could lead to a loss of this optimization. If ZFS is able to detect this situation, and attempt compression with the older version of Zstd, it can avoid the unexpected growth of snapshots of an Oracle database.

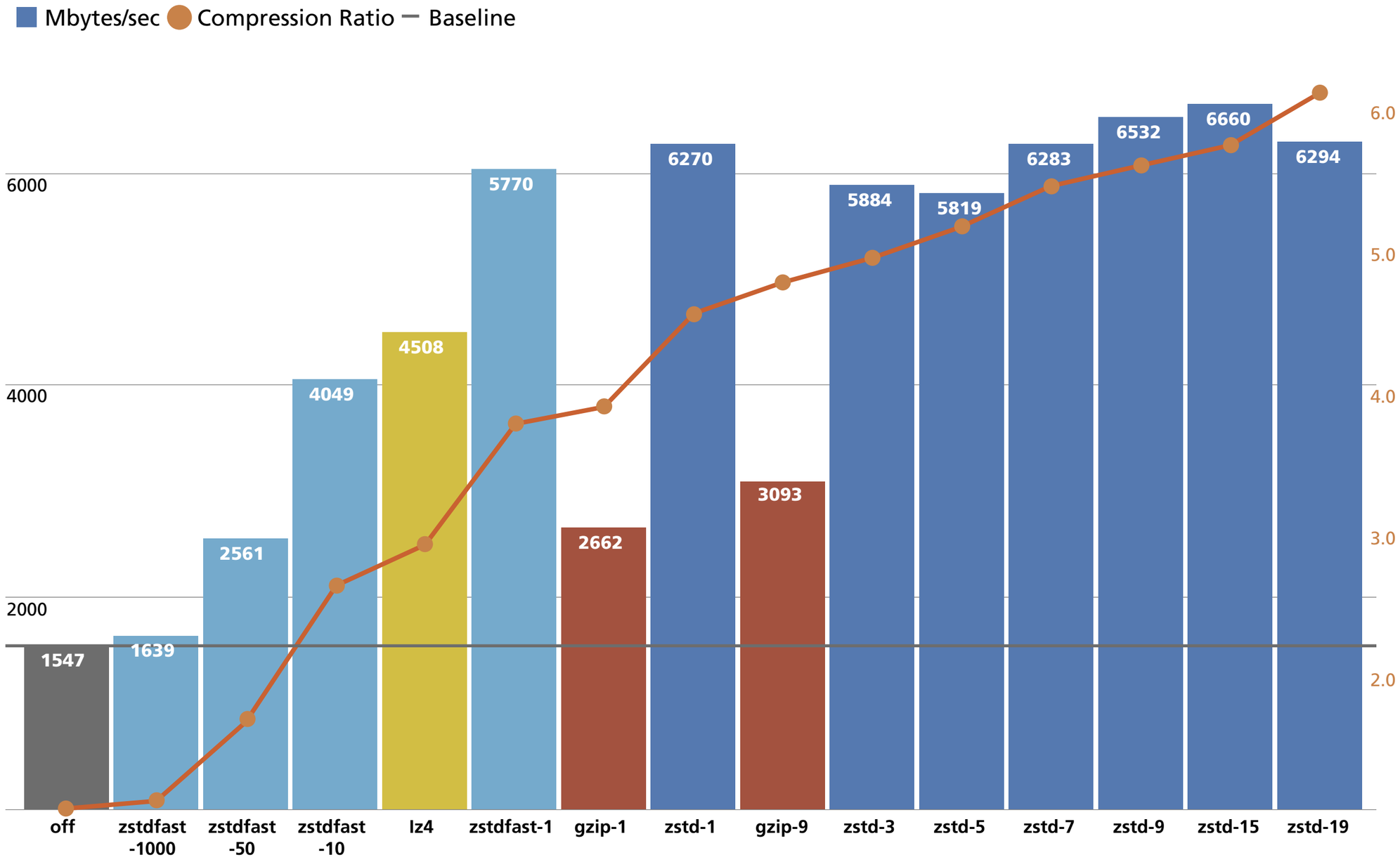
Where Zstd Shines

Zstandard provides a large selection of compression levels, allowing the storage administrator relatively fine-grained control over balancing performance and compression ratio. One of the main advantages of Zstd is that the decompression speed is independent of the compression level. For data that is written once but read many times, Zstd allows the use of the highest compression levels without a performance penalty. When writing large amounts of data, ZFS compresses each record individually, so it is able to take advantage of the many processor cores available on modern systems. Even when data is updated frequently, there are often performance gains that come from enabling compression. One of the biggest advantages comes from the compressed ARC feature--itself a recent improvement in ZFS. ZFS's Adaptive Replacement Cache (ARC) now caches the compressed version of the data in RAM and decompresses it each time it is requested. This allows the same amount of cache to store more (often much more) logical data and metadata, increasing the cache hit ratio, and improving performance for the most frequently and most recently accessed data. If upgrading from LZ4 to Zstd increases the on-disk compression ratio, those gains directly multiply the efficacy of every byte in the compressed ARC.

In the chart below, we compare storing a large uncompressed tarball of FreeBSD source code on ZFS using a variety of compression algorithms and levels. The test system used four striped SATA SSDs, the read speed without compression was limited by the available throughput of the underlying storage devices to around 1.5 GB/s, however, as the compression ratio of the data goes up, the read speed generally increases as well, since the limiting factor is still how fast the compressed data can be brought in from the underlying storage. Compared to gzip, Zstd decompresses much faster, and wastes few of these gains as it does not generally require more CPU time in decompression.



Interestingly, using a larger ZFS "record size" allows even greater ratios. The reason for this is ZFS compresses each record independently, so record size has a large impact on the possible compression gains; the larger the record, the more optimal the compression dictionary. gzip-9 sees the compression ratio increase from 4.3x to 4.7x, it only gains a modest 8% additional throughput, while Zstd-9 boots its ratio from 4.9x to 5.5x and gains 28% more performance, reaching more than four times the throughput the hardware is capable of.



One thing to be aware of is that ZFS will not store a block compressed if the savings from compression do not result in the savings of at least one disk sector. For example, on a typical database filesystem, with a recordsize of 16 KB, if the compression ratio is 1.32x, resulting in the final block being 12.1 KB, it will still require the same four 4 KB sectors to be stored, so it will be less work to just store the data uncompressed. However, if the compression ratio is 1.34x, requiring 11.9 KB of storage space, this can be achieved with just three 4 KB sectors, so ZFS will use the compressed version. The compressionratio property of a dataset returns the average of all the records.

What's Next?

The integration of Zstd into ZFS has just begun and the future undoubtedly holds many improvements. Already, we have thoughts along these lines. For example, we expect using the advanced Zstd API to provide more hints about the maximum size of the input data could reduce memory usage and improve Zstd's ability to take advantage of "early abort," which we spoke of early in the article. There are likely a number of opportunities to optimize the way ZFS sets up and tears down Zstd compression contexts and to increase the reuse of these contexts with the Zstd reset API, which one would expect to significantly improve compression performance with small blocks.

Aside from continuing to optimize Zstd for ZFS, the next obvious evolution is to remove the

need for the user to decide what Zstd level is best (there are 40 options to choose from after all). Instead, we envision a user simply setting compress=zstd-auto and ZFS dynamically adapts in some sensible way. When using Zstd from the command line, to compress a stream being sent over the network, the user can specify—adapt=min=3,max=10 and Zstd will vary the compression level based on how quickly the network buffer is emptied. This ensures that the compression is not a bottleneck by lowering the compression level if the network has available bandwidth, or conversely, by increasing the time spent on compression if the network is not able to keep up with the current compression level.

In ZFS, this would likely be modelled on the amount of "dirty" data (data waiting to be compressed and written to disk). When new data is written to ZFS, it will be compressed with the maximum compression level. If the rate of incoming writes is too high for ZFS to keep up with the requested level of compression, which results in the amount of dirty data steadily increasing, the compression level would lower incrementally, ideally settling on the maximum level that does not limit throughput. As always, the ZFS philosophy is to make sensible use of system resources while minimizing the need for adjustment and tweaking by the user.

Conclusion

Zstd support shipped as part of the recently released OpenZFS 2.0, which is available as replacement for the base ZFS in FreeBSD 12.2 via the sysutils/openzfs package and is integrated into the FreeBSD 13.0 development branch.

When new data is written to ZFS, it will be compressed with the maximum compression level.

I want to give a special thanks to everyone at the FreeBSD Foundation for the grant that made it possible to get this long-running project finished and merged in time for OpenZFS 2.0. Thanks also to Sebastian Gottschall, Kjeld Schouten-Lebbing, and Michael Niewöhner who did the Linux port, including the additional kmem compatibility code, and creating most of the tests included in the final patch. I also want to thank the team that worked to integrate FreeBSD support into the upstream OpenZFS repo, and everyone at the OpenZFS project. Lastly, my thanks also go out to everyone who tested and reviewed the many versions of the patches over the years until it was finally committed.

ALLAN JUDE is VP of Engineering at Klara Inc., a global FreeBSD Professional Services and Support company. He also hosts the premier weekly BSD podcast, BSDNow.tv and served on the FreeBSD Core team from 2016 to 2020. He is the co-author of *"FreeBSD Mastery: ZFS"* and *"FreeBSD Mastery: Advanced ZFS"* with Michael W. Lucas.