NAME

zfs - tuning of the ZFS kernel module

DESCRIPTION

The ZFS module supports these parameters:

dbuf_cache_max_bytes=**UINT64_MAX**B (u64)

Maximum size in bytes of the dbuf cache. The target size is determined by the MIN versus 1/2^**dbuf_cache_shift** (1/32nd) of the target ARC size. The behavior of the dbuf cache and its associated settings can be observed via the */proc/spl/kstat/zfs/dbufstats* kstat.

dbuf_metadata_cache_max_bytes=**UINT64_MAX**B (u64)

Maximum size in bytes of the metadata dbuf cache. The target size is determined by the MIN versus 1/2^**dbuf_metadata_cache_shift** (1/64th) of the target ARC size. The behavior of the metadata dbuf cache and its associated settings can be observed via the */proc/spl/kstat/zfs/dbufstats* kstat.

dbuf_cache_hiwater_pct=**10**% (uint)

The percentage over **dbuf_cache_max_bytes** when dbufs must be evicted directly.

dbuf_cache_lowater_pct=**10**% (uint)

The percentage below **dbuf_cache_max_bytes** when the evict thread stops evicting dbufs.

dbuf_cache_shift=**5** (uint)

Set the size of the dbuf cache (**dbuf_cache_max_bytes**) to a log2 fraction of the target ARC size.

dbuf_metadata_cache_shift=**6** (uint)

Set the size of the dbuf metadata cache (**dbuf_metadata_cache_max_bytes**) to a log2 fraction of the target ARC size.

dbuf_mutex_cache_shift=**0** (uint)

Set the size of the mutex array for the dbuf cache. When set to **0** the array is dynamically sized based on total system memory.

dmu_object_alloc_chunk_shift=**7** (128) (uint)

dnode slots allocated in a single operation as a power of 2. The default value minimizes lock contention for the bulk operation performed.

dmu_prefetch_max=**134217728**B (128 MiB) (uint)

Limit the amount we can prefetch with one call to this amount in bytes. This helps to limit the amount of memory that can be used by prefetching.

ignore_hole_birth (int)
Alias for **send_holes_without_birth_time**.

l2arc_feed_again=**1**|0 (int)
Turbo L2ARC warm-up. When the L2ARC is cold the fill interval will be set as fast as possible.

l2arc_feed_min_ms=**200** (u64)
Min feed interval in milliseconds. Requires **l2arc_feed_again**=*1* and only applicable in related situations.

l2arc_feed_secs=**1** (u64)
Seconds between L2ARC writing.

l2arc_headroom=**2** (u64)
How far through the ARC lists to search for L2ARC cacheable content, expressed as a multiplier of **l2arc_write_max**. ARC persistence across reboots can be achieved with persistent L2ARC by setting this parameter to **0**, allowing the full length of ARC lists to be searched for cacheable content.

l2arc_headroom_boost=**200**% (u64)
Scales **l2arc_headroom** by this percentage when L2ARC contents are being successfully compressed before writing. A value of **100** disables this feature.

l2arc_exclude_special=**0**|1 (int)
Controls whether buffers present on special vdevs are eligible for caching into L2ARC. If set to 1, exclude dbufs on special vdevs from being cached to L2ARC.

l2arc_mfuonly=**0**|1|2 (int)
Controls whether only MFU metadata and data are cached from ARC into L2ARC. This may be desired to avoid wasting space on L2ARC when reading/writing large amounts of data that are not expected to be accessed more than once.

The default is 0, meaning both MRU and MFU data and metadata are cached. When turning off this feature (setting it to 0), some MRU buffers will still be present in ARC and eventually cached on L2ARC. If **l2arc_noprefetch**=**0**, some prefetched buffers will be cached to L2ARC, and those might later transition to MRU, in which case the **l2arc_mru_asize** arcstat will not be

0.

Setting it to 1 means to L2 cache only MFU data and metadata.

Setting it to 2 means to L2 cache all metadata (MRU+MFU) but only MFU data (ie: MRU data are not cached). This can be the right setting to cache as much metadata as possible even when having high data turnover.

Regardless of **l2arc_noprefetch**, some MFU buffers might be evicted from ARC, accessed later on as prefetches and transition to MRU as prefetches. If accessed again they are counted as MRU and the **l2arc_mru_asize** arcstat will not be **0**.

The ARC status of L2ARC buffers when they were first cached in L2ARC can be seen in the **l2arc_mru_asize**, **l2arc_mfu_asize**, and **l2arc_prefetch_asize** arcstats when importing the pool or onlining a cache device if persistent L2ARC is enabled.

The **evict_l2_eligible_mru** arcstat does not take into account if this option is enabled as the information provided by the **evict_l2_eligible_m[rf]u** arcstats can be used to decide if toggling this option is appropriate for the current workload.

l2arc_meta_percent=**33**% (uint)

Percent of ARC size allowed for L2ARC-only headers. Since L2ARC buffers are not evicted on memory pressure, too many headers on a system with an irrationally large L2ARC can render it slow or unusable. This parameter limits L2ARC writes and rebuilds to achieve the target.

l2arc_trim_ahead=**0**% (u64)

Trims ahead of the current write size (**l2arc_write_max**) on L2ARC devices by this percentage of write size if we have filled the device. If set to **100** we TRIM twice the space required to accommodate upcoming writes. A minimum of **64 MiB** will be trimmed. It also enables TRIM of the whole L2ARC device upon creation or addition to an existing pool or if the header of the device is invalid upon importing a pool or onlining a cache device. A value of **0** disables TRIM on L2ARC altogether and is the default as it can put significant stress on the underlying storage devices. This will vary depending of how well the specific device handles these commands.

l2arc_noprefetch=**1**|0 (int)

Do not write buffers to L2ARC if they were prefetched but not used by applications. In case there are prefetched buffers in L2ARC and this option is later set, we do not read the prefetched buffers from L2ARC. Unsetting this option is useful for caching sequential reads from the disks to L2ARC and serve those reads from L2ARC later on. This may be beneficial in case the

L2ARC device is significantly faster in sequential reads than the disks of the pool.

Use **1** to disable and **0** to enable caching/reading prefetches to/from L2ARC.

l2arc_norw=**0**|1 (int)
No reads during writes.

l2arc_write_boost=**8388608**B (8 MiB) (u64)
Cold L2ARC devices will have **l2arc_write_max** increased by this amount while they remain cold.

l2arc_write_max=**8388608**B (8 MiB) (u64)
Max write bytes per interval.

l2arc_rebuild_enabled=**1**|0 (int)
Rebuild the L2ARC when importing a pool (persistent L2ARC). This can be disabled if there are problems importing a pool or attaching an L2ARC device (e.g. the L2ARC device is slow in reading stored log metadata, or the metadata has become somehow fragmented/unusable).

l2arc_rebuild_blocks_min_l2size=**1073741824**B (1 GiB) (u64)
Mininum size of an L2ARC device required in order to write log blocks in it. The log blocks are used upon importing the pool to rebuild the persistent L2ARC.

For L2ARC devices less than 1 GiB, the amount of data **l2arc_evict**() evicts is significant compared to the amount of restored L2ARC data. In this case, do not write log blocks in L2ARC in order not to waste space.

metaslab_aliquot=**1048576**B (1 MiB) (u64)
Metaslab granularity, in bytes. This is roughly similar to what would be referred to as the "stripe size" in traditional RAID arrays. In normal operation, ZFS will try to write this amount of data to each disk before moving on to the next top-level vdev.

metaslab_bias_enabled=**1**|0 (int)
Enable metaslab group biasing based on their vdevs' over- or under-utilization relative to the pool.

metaslab_force_ganging=**16777217**B (16 MiB + 1 B) (u64)
Make some blocks above a certain size be gang blocks. This option is used by the test suite to facilitate testing.

metaslab_force_ganging_pct=**3**% (uint)

For blocks that could be forced to be a gang block (due to **metaslab_force_ganging**), force this many of them to be gang blocks.

brt_zap_prefetch=**1**|0 (int)

Controls prefetching BRT records for blocks which are going to be cloned.

brt_zap_default_bs=**12** (4 KiB) (int)

Default BRT ZAP data block size as a power of 2. Note that changing this after creating a BRT on the pool will not affect existing BRTs, only newly created ones.

brt_zap_default_ibs=**12** (4 KiB) (int)

Default BRT ZAP indirect block size as a power of 2. Note that changing this after creating a BRT on the pool will not affect existing BRTs, only newly created ones.

ddt_zap_default_bs=**15** (32 KiB) (int)

Default DDT ZAP data block size as a power of 2. Note that changing this after creating a DDT on the pool will not affect existing DDTs, only newly created ones.

ddt_zap_default_ibs=**15** (32 KiB) (int)

Default DDT ZAP indirect block size as a power of 2. Note that changing this after creating a DDT on the pool will not affect existing DDTs, only newly created ones.

zfs_default_bs=**9** (512 B) (int)

Default dnode block size as a power of 2.

zfs_default_ibs=**17** (128 KiB) (int)

Default dnode indirect block size as a power of 2.

zfs_history_output_max=**1048576**B (1 MiB) (u64)

When attempting to log an output nvlist of an ioctl in the on-disk history, the output will not be stored if it is larger than this size (in bytes). This must be less than **DMU_MAX_ACCESS** (64 MiB). This applies primarily to **zfs_ioc_channel_program**() (cf. zfs-program(8)).

zfs_keep_log_spacemaps_at_export=**0**|1 (int)

Prevent log spacemaps from being destroyed during pool exports and destroys.

zfs_metaslab_segment_weight_enabled=**1**|0 (int)

Enable/disable segment-based metaslab selection.

zfs_metaslab_switch_threshold=**2** (int)

When using segment-based metaslab selection, continue allocating from the active metaslab until this option’s worth of buckets have been exhausted.

metaslab_debug_load=**0**|1 (int)

Load all metaslabs during pool import.

metaslab_debug_unload=**0**|1 (int)

Prevent metaslabs from being unloaded.

metaslab_fragmentation_factor_enabled=**1**|0 (int)

Enable use of the fragmentation metric in computing metaslab weights.

metaslab_df_max_search=**16777216**B (16 MiB) (uint)

Maximum distance to search forward from the last offset. Without this limit, fragmented pools can see *>100‘000* iterations and **metaslab_block_picker**() becomes the performance limiting factor on high-performance storage.

With the default setting of **16 MiB**, we typically see less than *500* iterations, even with very fragmented **ashift**=**9** pools. The maximum number of iterations possible is **metaslab_df_max_search / 2^(ashift+1)**. With the default setting of **16 MiB** this is *16*1024* (with **ashift**=**9**) or *2*1024* (with **ashift**=**12**).

metaslab_df_use_largest_segment=**0**|1 (int)

If not searching forward (due to **metaslab_df_max_search**, **metaslab_df_free_pct**, or **metaslab_df_alloc_threshold**), this tunable controls which segment is used. If set, we will use the largest free segment. If unset, we will use a segment of at least the requested size.

zfs_metaslab_max_size_cache_sec=**3600**s (1 hour) (u64)

When we unload a metaslab, we cache the size of the largest free chunk. We use that cached size to determine whether or not to load a metaslab for a given allocation. As more frees accumulate in that metaslab while it’s unloaded, the cached max size becomes less and less accurate. After a number of seconds controlled by this tunable, we stop considering the cached max size and start considering only the histogram instead.

zfs_metaslab_mem_limit=**25**% (uint)

When we are loading a new metaslab, we check the amount of memory being used to store metaslab range trees. If it is over a threshold, we attempt to unload the least recently used metaslab to prevent the system from clogging all of its memory with range trees. This tunable sets the percentage of total system memory that is the threshold.

zfs_metaslab_try_hard_before_gang=0|1 (int)

If unset, we will first try normal allocation.
If that fails then we will do a gang allocation.
If that fails then we will do a "try hard" gang allocation.
If that fails then we will have a multi-layer gang block.

If set, we will first try normal allocation.
If that fails then we will do a "try hard" allocation.
If that fails we will do a gang allocation.
If that fails we will do a "try hard" gang allocation.
If that fails then we will have a multi-layer gang block.

zfs_metaslab_find_max_tries=100 (uint)

When not trying hard, we only consider this number of the best metaslabs. This improves performance, especially when there are many metaslabs per vdev and the allocation can’t actually be satisfied (so we would otherwise iterate all metaslabs).

zfs_vdev_default_ms_count=200 (uint)

When a vdev is added, target this number of metaslabs per top-level vdev.

zfs_vdev_default_ms_shift=29 (512 MiB) (uint)

Default lower limit for metaslab size.

zfs_vdev_max_ms_shift=34 (16 GiB) (uint)

Default upper limit for metaslab size.

zfs_vdev_max_auto_ashift=14 (uint)

Maximum ashift used when optimizing for logical -> physical sector size on new top-level vdevs. May be increased up to **ASHIFT_MAX** (16), but this may negatively impact pool space efficiency.

zfs_vdev_min_auto_ashift=ASHIFT_MIN (9) (uint)

Minimum ashift used when creating new top-level vdevs.

zfs_vdev_min_ms_count=16 (uint)

Minimum number of metaslabs to create in a top-level vdev.

vdev_validate_skip=0|1 (int)

Skip label validation steps during pool import. Changing is not recommended unless you know what you’re doing and are recovering a damaged label.

zfs_vdev_ms_count_limit=**131072** (128k) (uint)
Practical upper limit of total metaslabs per top-level vdev.

metaslab_preload_enabled=**1**|0 (int)
Enable metaslab group preloading.

metaslab_preload_limit=**10** (uint)
Maximum number of metaslabs per group to preload

metaslab_preload_pct=**50** (uint)
Percentage of CPUs to run a metaslab preload taskq

metaslab_lba_weighting_enabled=**1**|0 (int)
Give more weight to metaslabs with lower LBAs, assuming they have greater bandwidth, as is typically the case on a modern constant angular velocity disk drive.

metaslab_unload_delay=**32** (uint)
After a metaslab is used, we keep it loaded for this many TXGs, to attempt to reduce unnecessary reloading. Note that both this many TXGs and **metaslab_unload_delay_ms** milliseconds must pass before unloading will occur.

metaslab_unload_delay_ms=**600000**ms (10 min) (uint)
After a metaslab is used, we keep it loaded for this many milliseconds, to attempt to reduce unnecessary reloading. Note, that both this many milliseconds and **metaslab_unload_delay** TXGs must pass before unloading will occur.

reference_history=**3** (uint)
Maximum reference holders being tracked when reference_tracking_enable is active.

reference_tracking_enable=**0**|1 (int)
Track reference holders to **refcount_t** objects (debug builds only).

send_holes_without_birth_time=**1**|0 (int)
When set, the **hole_birth** optimization will not be used, and all holes will always be sent during a **zfs send**. This is useful if you suspect your datasets are affected by a bug in **hole_birth**.

spa_config_path=*/etc/zfs/zpool.cache* (charp)
SPA config file.

spa_asize_inflation=**24** (uint)

Multiplication factor used to estimate actual disk consumption from the size of data being written. The default value is a worst case estimate, but lower values may be valid for a given pool depending on its configuration. Pool administrators who understand the factors involved may wish to specify a more realistic inflation factor, particularly if they operate close to quota or capacity limits.

spa_load_print_vdev_tree=**0**|1 (int)
Whether to print the vdev tree in the debugging message buffer during pool import.

spa_load_verify_data=**1**|0 (int)
Whether to traverse data blocks during an "extreme rewind" (**-X**) import.

An extreme rewind import normally performs a full traversal of all blocks in the pool for verification. If this parameter is unset, the traversal skips non-metadata blocks. It can be toggled once the import has started to stop or start the traversal of non-metadata blocks.

spa_load_verify_metadata=**1**|0 (int)
Whether to traverse blocks during an "extreme rewind" (**-X**) pool import.

An extreme rewind import normally performs a full traversal of all blocks in the pool for verification. If this parameter is unset, the traversal is not performed. It can be toggled once the import has started to stop or start the traversal.

spa_load_verify_shift=**4** (1/16th) (uint)
Sets the maximum number of bytes to consume during pool import to the log2 fraction of the target ARC size.

spa_slop_shift=**5** (1/32nd) (int)
Normally, we don’t allow the last **3.2%** (**1/2^spa_slop_shift**) of space in the pool to be consumed. This ensures that we don’t run the pool completely out of space, due to unaccounted changes (e.g. to the MOS). It also limits the worst-case time to allocate space. If we have less than this amount of free space, most ZPL operations (e.g. write, create) will return **ENOSPC**.

spa_upgrade_errlog_limit=**0** (uint)
Limits the number of on-disk error log entries that will be converted to the new format when enabling the **head_errlog** feature. The default is to convert all log entries.

vdev_removal_max_span=**32768**B (32 KiB) (uint)
During top-level vdev removal, chunks of data are copied from the vdev which may include free space in order to trade bandwidth for IOPS. This parameter determines the maximum span

of free space, in bytes, which will be included as "unnecessary" data in a chunk of copied data.

The default value here was chosen to align with **zfs_vdev_read_gap_limit**, which is a similar concept when doing regular reads (but there's no reason it has to be the same).

vdev_file_logical_ashift=**9** (512 B) (u64)
Logical ashift for file-based devices.

vdev_file_physical_ashift=**9** (512 B) (u64)
Physical ashift for file-based devices.

zap_iterate_prefetch=**1**|0 (int)
If set, when we start iterating over a ZAP object, prefetch the entire object (all leaf blocks). However, this is limited by **dmu_prefetch_max**.

zap_micro_max_size=**131072**B (128 KiB) (int)
Maximum micro ZAP size. A micro ZAP is upgraded to a fat ZAP, once it grows beyond the specified size.

zfetch_hole_shift=**2** (uint)
Log2 fraction of holes in speculative prefetch stream allowed for it to proceed.

zfetch_min_distance=**4194304**B (4 MiB) (uint)
Min bytes to prefetch per stream. Prefetch distance starts from the demand access size and quickly grows to this value, doubling on each hit. After that it may grow further by 1/8 per hit, but only if some prefetch since last time haven't completed in time to satisfy demand request, i.e. prefetch depth didn't cover the read latency or the pool got saturated.

zfetch_max_distance=**67108864**B (64 MiB) (uint)
Max bytes to prefetch per stream.

zfetch_max_idistance=**67108864**B (64 MiB) (uint)
Max bytes to prefetch indirects for per stream.

zfetch_max_reorder=**16777216**B (16 MiB) (uint)
Requests within this byte distance from the current prefetch stream position are considered parts of the stream, reordered due to parallel processing. Such requests do not advance the stream position immediately unless **zfetch_hole_shift** fill threshold is reached, but saved to fill holes in the stream later.

zfetch_max_streams=**8** (uint)
Max number of streams per zfetch (prefetch streams per file).

zfetch_min_sec_reap=**1** (uint)
Min time before inactive prefetch stream can be reclaimed

zfetch_max_sec_reap=**2** (uint)
Max time before inactive prefetch stream can be deleted

zfs_abd_scatter_enabled=**1**|0 (int)
Enables ARC from using scatter/gather lists and forces all allocations to be linear in kernel memory. Disabling can improve performance in some code paths at the expense of fragmented kernel memory.

zfs_abd_scatter_max_order=**MAX_ORDER-1** (uint)
Maximum number of consecutive memory pages allocated in a single block for scatter/gather lists.

The value of **MAX_ORDER** depends on kernel configuration.

zfs_abd_scatter_min_size=**1536**B (1.5 KiB) (uint)
This is the minimum allocation size that will use scatter (page-based) ABDs. Smaller allocations will use linear ABDs.

zfs_arc_dnode_limit=**0**B (u64)
When the number of bytes consumed by dnodes in the ARC exceeds this number of bytes, try to unpin some of it in response to demand for non-metadata. This value acts as a ceiling to the amount of dnode metadata, and defaults to **0**, which indicates that a percent which is based on **zfs_arc_dnode_limit_percent** of the ARC meta buffers that may be used for dnodes.

zfs_arc_dnode_limit_percent=**10**% (u64)
Percentage that can be consumed by dnodes of ARC meta buffers.

See also **zfs_arc_dnode_limit**, which serves a similar purpose but has a higher priority if nonzero.

zfs_arc_dnode_reduce_percent=**10**% (u64)
Percentage of ARC dnodes to try to scan in response to demand for non-metadata when the number of bytes consumed by dnodes exceeds **zfs_arc_dnode_limit**.

zfs_arc_average_blocksize=**8192**B (8 KiB) (uint)

The ARC's buffer hash table is sized based on the assumption of an average block size of this value. This works out to roughly 1 MiB of hash table per 1 GiB of physical memory with 8-byte pointers. For configurations with a known larger average block size, this value can be increased to reduce the memory footprint.

zfs_arc_eviction_pct=**200**% (uint)

When **arc_is_overflowing**(), **arc_get_data_impl**() waits for this percent of the requested amount of data to be evicted. For example, by default, for every *2 KiB* that's evicted, *1 KiB* of it may be "reused" by a new allocation. Since this is above **100**%, it ensures that progress is made towards getting **arc_size** under **arc_c**. Since this is finite, it ensures that allocations can still happen, even during the potentially long time that **arc_size** is more than **arc_c**.

zfs_arc_evict_batch_limit=**10** (uint)

Number ARC headers to evict per sub-list before proceeding to another sub-list. This batch-style operation prevents entire sub-lists from being evicted at once but comes at a cost of additional unlocking and locking.

zfs_arc_grow_retry=**0**s (uint)

If set to a non zero value, it will replace the **arc_grow_retry** value with this value. The **arc_grow_retry** value (default **5**s) is the number of seconds the ARC will wait before trying to resume growth after a memory pressure event.

zfs_arc_lotsfree_percent=**10**% (int)

Throttle I/O when free system memory drops below this percentage of total system memory. Setting this value to **0** will disable the throttle.

zfs_arc_max=**0**B (u64)

Max size of ARC in bytes. If **0**, then the max size of ARC is determined by the amount of system memory installed. Under Linux, half of system memory will be used as the limit. Under FreeBSD, the larger of **all_system_memory - 1 GiB** and **5/8** x **all_system_memory** will be used as the limit. This value must be at least **67108864**B (64 MiB).

This value can be changed dynamically, with some caveats. It cannot be set back to **0** while running, and reducing it below the current ARC size will not cause the ARC to shrink without memory pressure to induce shrinking.

zfs_arc_meta_balance=**500** (uint)

Balance between metadata and data on ghost hits. Values above 100 increase metadata caching by proportionally reducing effect of ghost data hits on target data/metadata rate.

zfs_arc_min=**0**B (u64)

Min size of ARC in bytes. If set to **0**, **arc_c_min** will default to consuming the larger of **32 MiB** and **all_system_memory** / **32**.

zfs_arc_min_prefetch_ms=**0**ms(==1s) (uint)

Minimum time prefetched blocks are locked in the ARC.

zfs_arc_min_prescient_prefetch_ms=**0**ms(==6s) (uint)

Minimum time "prescient prefetched" blocks are locked in the ARC. These blocks are meant to be prefetched fairly aggressively ahead of the code that may use them.

zfs_arc_prune_task_threads=**1** (int)

Number of arc_prune threads. FreeBSD does not need more than one. Linux may theoretically use one per mount point up to number of CPUs, but that was not proven to be useful.

zfs_max_missing_tvds=**0** (int)

Number of missing top-level vdevs which will be allowed during pool import (only in read-only mode).

zfs_max_nvlist_src_size= **0** (u64)

Maximum size in bytes allowed to be passed as **zc_nvlist_src_size** for ioctls on */dev/zfs*. This prevents a user from causing the kernel to allocate an excessive amount of memory. When the limit is exceeded, the ioctl fails with **EINVAL** and a description of the error is sent to the *zfs-dbgmsg* log. This parameter should not need to be touched under normal circumstances. If **0**, equivalent to a quarter of the user-wired memory limit under FreeBSD and to **134217728**B (128 MiB) under Linux.

zfs_multilist_num_sublists=**0** (uint)

To allow more fine-grained locking, each ARC state contains a series of lists for both data and metadata objects. Locking is performed at the level of these "sub-lists". This parameters controls the number of sub-lists per ARC state, and also applies to other uses of the multilist data structure.

If **0**, equivalent to the greater of the number of online CPUs and **4**.

zfs_arc_overflow_shift=**8** (int)

The ARC size is considered to be overflowing if it exceeds the current ARC target size (**arc_c**) by thresholds determined by this parameter. Exceeding by (**arc_c** >> **zfs_arc_overflow_shift**) / **2** starts ARC reclamation process. If that appears insufficient, exceeding by (**arc_c** >> **zfs_arc_overflow_shift**) x **1.5** blocks new buffer allocation until the reclaim thread catches up.

Started reclamation process continues till ARC size returns below the target size.

The default value of **8** causes the ARC to start reclamation if it exceeds the target size by *0.2%* of the target size, and block allocations by *0.6%*.

zfs_arc_shrink_shift=**0** (uint)

If nonzero, this will update **arc_shrink_shift** (default **7**) with the new value.

zfs_arc_pc_percent=**0**% (off) (uint)

Percent of pagecache to reclaim ARC to.

This tunable allows the ZFS ARC to play more nicely with the kernel's LRU pagecache. It can guarantee that the ARC size won't collapse under scanning pressure on the pagecache, yet still allows the ARC to be reclaimed down to **zfs_arc_min** if necessary. This value is specified as percent of pagecache size (as measured by **NR_FILE_PAGES**), where that percent may exceed **100**. This only operates during memory pressure/reclaim.

zfs_arc_shrinker_limit=**10000** (int)

This is a limit on how many pages the ARC shrinker makes available for eviction in response to one page allocation attempt. Note that in practice, the kernel's shrinker can ask us to evict up to about four times this for one allocation attempt.

The default limit of **10000** (in practice, *160 MiB* per allocation attempt with 4 KiB pages) limits the amount of time spent attempting to reclaim ARC memory to less than 100 ms per allocation attempt, even with a small average compressed block size of ~8 KiB.

The parameter can be set to 0 (zero) to disable the limit, and only applies on Linux.

zfs_arc_sys_free=**0**B (u64)

The target number of bytes the ARC should leave as free memory on the system. If zero, equivalent to the bigger of **512 KiB** and **all_system_memory/64**.

zfs_autoimport_disable=**1**|0 (int)

Disable pool import at module load by ignoring the cache file (**spa_config_path**).

zfs_checksum_events_per_second=**20**/s (uint)

Rate limit checksum events to this many per second. Note that this should not be set below the ZED thresholds (currently 10 checksums over 10 seconds) or else the daemon may not trigger any action.

zfs_commit_timeout_pct=**10**% (uint)

This controls the amount of time that a ZIL block (lwb) will remain "open" when it isn't "full", and it has a thread waiting for it to be committed to stable storage. The timeout is scaled based on a percentage of the last lwb latency to avoid significantly impacting the latency of each individual transaction record (itx).

zfs_condense_indirect_commit_entry_delay_ms=**0**ms (int)

Vdev indirection layer (used for device removal) sleeps for this many milliseconds during mapping generation. Intended for use with the test suite to throttle vdev removal speed.

zfs_condense_indirect_obsolete_pct=**25**% (uint)

Minimum percent of obsolete bytes in vdev mapping required to attempt to condense (see **zfs_condense_indirect_vdevs_enable**). Intended for use with the test suite to facilitate triggering condensing as needed.

zfs_condense_indirect_vdevs_enable=**1**|0 (int)

Enable condensing indirect vdev mappings. When set, attempt to condense indirect vdev mappings if the mapping uses more than **zfs_condense_min_mapping_bytes** bytes of memory and if the obsolete space map object uses more than **zfs_condense_max_obsolete_bytes** bytes on-disk. The condensing process is an attempt to save memory by removing obsolete mappings.

zfs_condense_max_obsolete_bytes=**1073741824**B (1 GiB) (u64)

Only attempt to condense indirect vdev mappings if the on-disk size of the obsolete space map object is greater than this number of bytes (see **zfs_condense_indirect_vdevs_enable**).

zfs_condense_min_mapping_bytes=**131072**B (128 KiB) (u64)

Minimum size vdev mapping to attempt to condense (see **zfs_condense_indirect_vdevs_enable**).

zfs_dbgmsg_enable=**1**|0 (int)

Internally ZFS keeps a small log to facilitate debugging. The log is enabled by default, and can be disabled by unsetting this option. The contents of the log can be accessed by reading */proc/spl/kstat/zfs/dbgmsg*. Writing **0** to the file clears the log.

This setting does not influence debug prints due to **zfs_flags**.

zfs_dbgmsg_maxsize=**4194304**B (4 MiB) (uint)

Maximum size of the internal ZFS debug log.

zfs_dbuf_state_index=0 (int)

Historically used for controlling what reporting was available under */proc/spl/kstat/zfs*. No effect.

zfs_deadman_enabled=**1**|0 (int)

When a pool sync operation takes longer than **zfs_deadman_synctime_ms**, or when an individual I/O operation takes longer than **zfs_deadman_ziotime_ms**, then the operation is considered to be "hung". If **zfs_deadman_enabled** is set, then the deadman behavior is invoked as described by **zfs_deadman_failmode**. By default, the deadman is enabled and set to **wait** which results in "hung" I/O operations only being logged. The deadman is automatically disabled when a pool gets suspended.

zfs_deadman_failmode=**wait** (charp)

Controls the failure behavior when the deadman detects a "hung" I/O operation. Valid values are:

wait Wait for a "hung" operation to complete. For each "hung" operation a "deadman" event will be posted describing that operation.

continue Attempt to recover from a "hung" operation by re-dispatching it to the I/O pipeline if possible.

panic Panic the system. This can be used to facilitate automatic fail-over to a properly configured fail-over partner.

zfs_deadman_checktime_ms=**60000**ms (1 min) (u64)

Check time in milliseconds. This defines the frequency at which we check for hung I/O requests and potentially invoke the **zfs_deadman_failmode** behavior.

zfs_deadman_synctime_ms=**600000**ms (10 min) (u64)

Interval in milliseconds after which the deadman is triggered and also the interval after which a pool sync operation is considered to be "hung". Once this limit is exceeded the deadman will be invoked every **zfs_deadman_checktime_ms** milliseconds until the pool sync completes.

zfs_deadman_ziotime_ms=**300000**ms (5 min) (u64)

Interval in milliseconds after which the deadman is triggered and an individual I/O operation is considered to be "hung". As long as the operation remains "hung", the deadman will be invoked every **zfs_deadman_checktime_ms** milliseconds until the operation completes.

zfs_dedup_prefetch=**0**|1 (int)

Enable prefetching dedup-ed blocks which are going to be freed.

zfs_delay_min_dirty_percent=**60**% (uint)

Start to delay each transaction once there is this amount of dirty data, expressed as a percentage of **zfs_dirty_data_max**. This value should be at least **zfs_vdev_async_write_active_max_dirty_percent**. See *ZFS TRANSACTION DELAY*.

zfs_delay_scale=**500000** (int)

This controls how quickly the transaction delay approaches infinity. Larger values cause longer delays for a given amount of dirty data.

For the smoothest delay, this value should be about 1 billion divided by the maximum number of operations per second. This will smoothly handle between ten times and a tenth of this number. See *ZFS TRANSACTION DELAY*.

zfs_delay_scale x **zfs_dirty_data_max** *must* be smaller than **2^64**.

zfs_disable_ivset_guid_check=**0**|1 (int)

Disables requirement for IVset GUIDs to be present and match when doing a raw receive of encrypted datasets. Intended for users whose pools were created with OpenZFS pre-release versions and now have compatibility issues.

zfs_key_max_salt_uses=**400000000** (4*10^8) (ulong)

Maximum number of uses of a single salt value before generating a new one for encrypted datasets. The default value is also the maximum.

zfs_object_mutex_size=**64** (uint)

Size of the znode hashtable used for holds.

Due to the need to hold locks on objects that may not exist yet, kernel mutexes are not created per-object and instead a hashtable is used where collisions will result in objects waiting when there is not actually contention on the same object.

zfs_slow_io_events_per_second=**20**/s (int)

Rate limit delay and deadman zevents (which report slow I/O operations) to this many per second.

zfs_unflushed_max_mem_amt=**1073741824**B (1 GiB) (u64)

Upper-bound limit for unflushed metadata changes to be held by the log spacemap in memory, in bytes.

zfs_unflushed_max_mem_ppm=**1000**ppm (0.1%) (u64)

Part of overall system memory that ZFS allows to be used for unflushed metadata changes by

the log spacemap, in millionths.

zfs_unflushed_log_block_max=**131072** (128k) (u64)

Describes the maximum number of log spacemap blocks allowed for each pool. The default value means that the space in all the log spacemaps can add up to no more than **131072** blocks (which means *16 GiB* of logical space before compression and ditto blocks, assuming that blocksize is *128 KiB*).

This tunable is important because it involves a trade-off between import time after an unclean export and the frequency of flushing metaslabs. The higher this number is, the more log blocks we allow when the pool is active which means that we flush metaslabs less often and thus decrease the number of I/O operations for spacemap updates per TXG. At the same time though, that means that in the event of an unclean export, there will be more log spacemap blocks for us to read, inducing overhead in the import time of the pool. The lower the number, the amount of flushing increases, destroying log blocks quicker as they become obsolete faster, which leaves less blocks to be read during import time after a crash.

Each log spacemap block existing during pool import leads to approximately one extra logical I/O issued. This is the reason why this tunable is exposed in terms of blocks rather than space used.

zfs_unflushed_log_block_min=**1000** (u64)

If the number of metaslabs is small and our incoming rate is high, we could get into a situation that we are flushing all our metaslabs every TXG. Thus we always allow at least this many log blocks.

zfs_unflushed_log_block_pct=**400**% (u64)

Tunable used to determine the number of blocks that can be used for the spacemap log, expressed as a percentage of the total number of unflushed metaslabs in the pool.

zfs_unflushed_log_txg_max=**1000** (u64)

Tunable limiting maximum time in TXGs any metaslab may remain unflushed. It effectively limits maximum number of unflushed per-TXG spacemap logs that need to be read after unclean pool export.

zfs_unlink_suspend_progress=**0**|1 (uint)

When enabled, files will not be asynchronously removed from the list of pending unlinks and the space they consume will be leaked. Once this option has been disabled and the dataset is remounted, the pending unlinks will be processed and the freed space returned to the pool. This option is used by the test suite.

zfs_delete_blocks=**20480** (ulong)

This is the used to define a large file for the purposes of deletion. Files containing more than **zfs_delete_blocks** will be deleted asynchronously, while smaller files are deleted synchronously. Decreasing this value will reduce the time spent in an unlink(2) system call, at the expense of a longer delay before the freed space is available. This only applies on Linux.

zfs_dirty_data_max= (int)

Determines the dirty space limit in bytes. Once this limit is exceeded, new writes are halted until space frees up. This parameter takes precedence over **zfs_dirty_data_max_percent**. See *ZFS TRANSACTION DELAY*.

Defaults to **physical_ram/10**, capped at **zfs_dirty_data_max_max**.

zfs_dirty_data_max_max= (int)

Maximum allowable value of **zfs_dirty_data_max**, expressed in bytes. This limit is only enforced at module load time, and will be ignored if **zfs_dirty_data_max** is later changed. This parameter takes precedence over **zfs_dirty_data_max_max_percent**. See *ZFS TRANSACTION DELAY*.

Defaults to **min(physical_ram/4, 4GiB)**, or **min(physical_ram/4, 1GiB)** for 32-bit systems.

zfs_dirty_data_max_max_percent=**25**% (uint)

Maximum allowable value of **zfs_dirty_data_max**, expressed as a percentage of physical RAM. This limit is only enforced at module load time, and will be ignored if **zfs_dirty_data_max** is later changed. The parameter **zfs_dirty_data_max_max** takes precedence over this one. See *ZFS TRANSACTION DELAY*.

zfs_dirty_data_max_percent=**10**% (uint)

Determines the dirty space limit, expressed as a percentage of all memory. Once this limit is exceeded, new writes are halted until space frees up. The parameter **zfs_dirty_data_max** takes precedence over this one. See *ZFS TRANSACTION DELAY*.

Subject to **zfs_dirty_data_max_max**.

zfs_dirty_data_sync_percent=**20**% (uint)

Start syncing out a transaction group if there’s at least this much dirty data (as a percentage of **zfs_dirty_data_max**). This should be less than **zfs_vdev_async_write_active_min_dirty_percent**.

zfs_wrlog_data_max= (int)

The upper limit of write-transaction zil log data size in bytes. Write operations are throttled when approaching the limit until log data is cleared out after transaction group sync. Because of some overhead, it should be set at least 2 times the size of **zfs_dirty_data_max** to prevent harming normal write throughput. It also should be smaller than the size of the slog device if slog is present.

Defaults to **zfs_dirty_data_max*2**

zfs_fallocate_reserve_percent=**110**% (uint)

Since ZFS is a copy-on-write filesystem with snapshots, blocks cannot be preallocated for a file in order to guarantee that later writes will not run out of space. Instead, fallocate(2) space preallocation only checks that sufficient space is currently available in the pool or the user's project quota allocation, and then creates a sparse file of the requested size. The requested space is multiplied by **zfs_fallocate_reserve_percent** to allow additional space for indirect blocks and other internal metadata. Setting this to **0** disables support for fallocate(2) and causes it to return **EOPNOTSUPP**.

zfs_fletcher_4_impl=**fastest** (string)

Select a fletcher 4 implementation.

Supported selectors are: **fastest**, **scalar**, **sse2**, **ssse3**, **avx2**, **avx512f**, **avx512bw**, and **aarch64_neon**. All except **fastest** and **scalar** require instruction set extensions to be available, and will only appear if ZFS detects that they are present at runtime. If multiple implementations of fletcher 4 are available, the **fastest** will be chosen using a micro benchmark. Selecting **scalar** results in the original CPU-based calculation being used. Selecting any option other than **fastest** or **scalar** results in vector instructions from the respective CPU instruction set being used.

zfs_bclone_enabled=**1**|0 (int)

Enable the experimental block cloning feature. If this setting is 0, then even if feature@block_cloning is enabled, attempts to clone blocks will act as though the feature is disabled.

zfs_bclone_wait_dirty=**0**|1 (int)

When set to 1 the FICLONE and FICLONERANGE ioctls wait for dirty data to be written to disk. This allows the clone operation to reliably succeed when a file is modified and then immediately cloned. For small files this may be slower than making a copy of the file. Therefore, this setting defaults to 0 which causes a clone operation to immediately fail when encountering a dirty block.

zfs_blake3_impl=fastest (string)
Select a BLAKE3 implementation.

Supported selectors are: **cycle**, **fastest**, **generic**, **sse2**, **sse41**, **avx2**, **avx512**. All except **cycle**, **fastest** and **generic** require instruction set extensions to be available, and will only appear if ZFS detects that they are present at runtime. If multiple implementations of BLAKE3 are available, the **fastest will be chosen using a micro benchmark. You can see the** benchmark results by reading this kstat file: */proc/spl/kstat/zfs/chksum_bench*.

zfs_free_bpobj_enabled=1|0 (int)
Enable/disable the processing of the free_bpobj object.

zfs_async_block_max_blocks=UINT64_MAX (unlimited) (u64)
Maximum number of blocks freed in a single TXG.

zfs_max_async_dedup_frees=100000 (10^5) (u64)
Maximum number of dedup blocks freed in a single TXG.

zfs_vdev_async_read_max_active=3 (uint)
Maximum asynchronous read I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_async_read_min_active=1 (uint)
Minimum asynchronous read I/O operation active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_async_write_active_max_dirty_percent=60% (uint)
When the pool has more than this much dirty data, use **zfs_vdev_async_write_max_active** to limit active async writes. If the dirty data is between the minimum and maximum, the active I/O limit is linearly interpolated. See *ZFS I/O SCHEDULER*.

zfs_vdev_async_write_active_min_dirty_percent=30% (uint)
When the pool has less than this much dirty data, use **zfs_vdev_async_write_min_active** to limit active async writes. If the dirty data is between the minimum and maximum, the active I/O limit is linearly interpolated. See *ZFS I/O SCHEDULER*.

zfs_vdev_async_write_max_active=10 (uint)
Maximum asynchronous write I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_async_write_min_active=2 (uint)

Minimum asynchronous write I/O operations active to each device. See *ZFS I/O SCHEDULER*.

Lower values are associated with better latency on rotational media but poorer resilver performance. The default value of **2** was chosen as a compromise. A value of **3** has been shown to improve resilver performance further at a cost of further increasing latency.

zfs_vdev_initializing_max_active=**1** (uint)
Maximum initializing I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_initializing_min_active=**1** (uint)
Minimum initializing I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_max_active=**1000** (uint)
The maximum number of I/O operations active to each device. Ideally, this will be at least the sum of each queue's **max_active**. See *ZFS I/O SCHEDULER*.

zfs_vdev_open_timeout_ms=**1000** (uint)
Timeout value to wait before determining a device is missing during import. This is helpful for transient missing paths due to links being briefly removed and recreated in response to udev events.

zfs_vdev_rebuild_max_active=**3** (uint)
Maximum sequential resilver I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_rebuild_min_active=**1** (uint)
Minimum sequential resilver I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_removal_max_active=**2** (uint)
Maximum removal I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_removal_min_active=**1** (uint)
Minimum removal I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_scrub_max_active=**2** (uint)
Maximum scrub I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_scrub_min_active=**1** (uint)
Minimum scrub I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_sync_read_max_active=**10** (uint)
Maximum synchronous read I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_sync_read_min_active=**10** (uint)
Minimum synchronous read I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_sync_write_max_active=**10** (uint)
Maximum synchronous write I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_sync_write_min_active=**10** (uint)
Minimum synchronous write I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_trim_max_active=**2** (uint)
Maximum trim/discard I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_trim_min_active=**1** (uint)
Minimum trim/discard I/O operations active to each device. See *ZFS I/O SCHEDULER*.

zfs_vdev_nia_delay=**5** (uint)
For non-interactive I/O (scrub, resilver, removal, initialize and rebuild), the number of concurrently-active I/O operations is limited to **zfs_*_min_active**, unless the vdev is "idle". When there are no interactive I/O operations active (synchronous or otherwise), and **zfs_vdev_nia_delay** operations have completed since the last interactive operation, then the vdev is considered to be "idle", and the number of concurrently-active non-interactive operations is increased to **zfs_*_max_active**. See *ZFS I/O SCHEDULER*.

zfs_vdev_nia_credit=**5** (uint)
Some HDDs tend to prioritize sequential I/O so strongly, that concurrent random I/O latency reaches several seconds. On some HDDs this happens even if sequential I/O operations are submitted one at a time, and so setting **zfs_*_max_active**= **1** does not help. To prevent non-interactive I/O, like scrub, from monopolizing the device, no more than **zfs_vdev_nia_credit operations can be sent** while there are outstanding incomplete interactive operations. This enforced wait ensures the HDD services the interactive I/O within a reasonable amount of time. See *ZFS I/O SCHEDULER*.

zfs_vdev_queue_depth_pct=**1000**% (uint)
Maximum number of queued allocations per top-level vdev expressed as a percentage of **zfs_vdev_async_write_max_active**, which allows the system to detect devices that are more capable of handling allocations and to allocate more blocks to those devices. This allows for dynamic allocation distribution when devices are imbalanced, as fuller devices will tend to be

slower than empty devices.

Also see **zio_dva_throttle_enabled**.

zfs_vdev_def_queue_depth=**32** (uint)

Default queue depth for each vdev IO allocator. Higher values allow for better coalescing of sequential writes before sending them to the disk, but can increase transaction commit times.

zfs_vdev_failfast_mask=**1** (uint)

Defines if the driver should retire on a given error type. The following options may be bitwise-ored together:

ValueName	Description
1	Device No driver retries on device errors
2	Transport No driver retries on transport errors.
4	Driver No driver retries on driver errors.

zfs_vdev_disk_max_segs=**0** (uint)

Maximum number of segments to add to a BIO (min 4). If this is higher than the maximum allowed by the device queue or the kernel itself, it will be clamped. Setting it to zero will cause the kernel's ideal size to be used. This parameter only applies on Linux. This parameter is ignored if **zfs_vdev_disk_classic**=**1**.

zfs_vdev_disk_classic=0|**1** (uint)

Controls the method used to submit IO to the Linux block layer (default **1 classic**)

If set to 1, the "classic" method is used. This is the method that has been in use since the earliest versions of ZFS-on-Linux. It has known issues with highly fragmented IO requests and is less efficient on many workloads, but it well known and well understood.

If set to 0, the "new" method is used. This method is available since 2.2.4 and should resolve all known issues and be far more efficient, but has not had as much testing. In the 2.2.x series, this parameter defaults to 1, to use the "classic" method.

It is not recommended that you change it except on advice from the OpenZFS developers. If you do change it, please also open a bug report describing why you did so, including the

workload involved and any error messages.

This parameter and the "classic" submission method will be removed in a future release of OpenZFS once we have total confidence in the new method.

This parameter only applies on Linux, and can only be set at module load time.

zfs_expire_snapshot=300s (int)
Time before expiring *.zfs/snapshot*.

zfs_admin_snapshot=0|1 (int)
Allow the creation, removal, or renaming of entries in the **.zfs/snapshot** directory to cause the creation, destruction, or renaming of snapshots. When enabled, this functionality works both locally and over NFS exports which have the *no_root_squash* option set.

zfs_flags=0 (int)
Set additional debugging flags. The following flags may be bitwise-ored together:

	Value	Name	Description
	1	ZFS_DEBUG_DPRINTF	Enable dprintf entries in the debug log.
*	2	ZFS_DEBUG_DBUF_VERIFY	Enable extra dbuf verifications.
*	4	ZFS_DEBUG_DNODE_VERIFY	Enable extra dnode verifications.
	8	ZFS_DEBUG_SNAPNAMES	Enable snapshot name verification.
*	16	ZFS_DEBUG_MODIFY	Check for illegally modified ARC buffers.
	64	ZFS_DEBUG_ZIO_FREE	Enable verification of block frees.
	128	ZFS_DEBUG_HISTOGRAM_VERIFY	Enable extra spacemap histogram verifications.
	256	ZFS_DEBUG_METASLAB_VERIFY	Verify space accounting on disk matches in-memory **range_trees**
	512	ZFS_DEBUG_SET_ERROR	Enable **SET_ERROR** and dprintf entries in the debug log.
	1024	ZFS_DEBUG_INDIRECT_REMAP	Verify split blocks created by device removal.
	2048	ZFS_DEBUG_TRIM	Verify TRIM ranges are always within the allocatable range

	tree.
4096ZFS_DEBUG_LOG_SPACEMAP	Verify that the log summary is consistent with the spacemap log and enable **zfs_dbgmsgs** for metaslab loading and flushing.

***** Requires debug build.

zfs_btree_verify_intensity=**0** (uint)

Enables btree verification. The following settings are culminative:

Value	Description
1	Verify height.
2	Verify pointers from children to parent.
3	Verify element counts.
4	Verify element order. (expensive)
***** 5	Verify unused memory is poisoned. (expensive)

***** Requires debug build.

zfs_free_leak_on_eio=**0**|1 (int)

If destroy encounters an **EIO** while reading metadata (e.g. indirect blocks), space referenced by the missing metadata can not be freed. Normally this causes the background destroy to become "stalled", as it is unable to make forward progress. While in this stalled state, all remaining space to free from the error-encountering filesystem is "temporarily leaked". Set this flag to cause it to ignore the **EIO**, permanently leak the space from indirect blocks that can not be read, and continue to free everything else that it can.

The default "stalling" behavior is useful if the storage partially fails (i.e. some but not all I/O operations fail), and then later recovers. In this case, we will be able to continue pool operations while it is partially failed, and when it recovers, we can continue to free the space, with no leaks. Note, however, that this case is actually fairly rare.

Typically pools either

1. fail completely (but perhaps temporarily, e.g. due to a top-level vdev going offline), or
2. have localized, permanent errors (e.g. disk returns the wrong data due to bit flip or firmware bug).

In the former case, this setting does not matter because the pool will be suspended and the sync thread will not be able to make forward progress regardless. In the latter, because the error is permanent, the best we can do is leak the minimum amount of space, which is what setting this flag will do. It is therefore reasonable for this flag to normally be set, but we chose the more conservative approach of not setting it, so that there is no possibility of leaking space in the "partial temporary" failure case.

zfs_free_min_time_ms=**1000**ms (1s) (uint)

During a **zfs destroy** operation using the **async_destroy** feature, a minimum of this much time will be spent working on freeing blocks per TXG.

zfs_obsolete_min_time_ms=**500**ms (uint)

Similar to **zfs_free_min_time_ms**, but for cleanup of old indirection records for removed vdevs.

zfs_immediate_write_sz=**32768**B (32 KiB) (s64)

Largest data block to write to the ZIL. Larger blocks will be treated as if the dataset being written to had the **logbias**=**throughput** property set.

zfs_initialize_value=**16045690984833335022** (0xDEADBEEFDEADBEEE) (u64)

Pattern written to vdev free space by zpool-initialize(8).

zfs_initialize_chunk_size=**1048576**B (1 MiB) (u64)

Size of writes used by zpool-initialize(8). This option is used by the test suite.

zfs_livelist_max_entries=**500000** (5*10^5) (u64)

The threshold size (in block pointers) at which we create a new sub-livelist. Larger sublists are more costly from a memory perspective but the fewer sublists there are, the lower the cost of insertion.

zfs_livelist_min_percent_shared=**75**% (int)

If the amount of shared space between a snapshot and its clone drops below this threshold, the clone turns off the livelist and reverts to the old deletion method. This is in place because livelists no long give us a benefit once a clone has been overwritten enough.

zfs_livelist_condense_new_alloc=**0** (int)

Incremented each time an extra ALLOC blkptr is added to a livelist entry while it is being condensed. This option is used by the test suite to track race conditions.

zfs_livelist_condense_sync_cancel=**0** (int)

Incremented each time livelist condensing is canceled while in **spa_livelist_condense_sync**(). This option is used by the test suite to track race conditions.

zfs_livelist_condense_sync_pause=**0**|1 (int)

When set, the livelist condense process pauses indefinitely before executing the synctask -- **spa_livelist_condense_sync**(). This option is used by the test suite to trigger race conditions.

zfs_livelist_condense_zthr_cancel=**0** (int)

Incremented each time livelist condensing is canceled while in **spa_livelist_condense_cb**(). This option is used by the test suite to track race conditions.

zfs_livelist_condense_zthr_pause=**0**|1 (int)

When set, the livelist condense process pauses indefinitely before executing the open context condensing work in **spa_livelist_condense_cb**(). This option is used by the test suite to trigger race conditions.

zfs_lua_max_instrlimit=**100000000** (10^8) (u64)

The maximum execution time limit that can be set for a ZFS channel program, specified as a number of Lua instructions.

zfs_lua_max_memlimit=**104857600** (100 MiB) (u64)

The maximum memory limit that can be set for a ZFS channel program, specified in bytes.

zfs_max_dataset_nesting=**50** (int)

The maximum depth of nested datasets. This value can be tuned temporarily to fix existing datasets that exceed the predefined limit.

zfs_max_log_walking=**5** (u64)

The number of past TXGs that the flushing algorithm of the log spacemap feature uses to estimate incoming log blocks.

zfs_max_logsm_summary_length=**10** (u64)

Maximum number of rows allowed in the summary of the spacemap log.

zfs_max_recordsize=**16777216** (16 MiB) (uint)

We currently support block sizes from *512* (512 B) to *16777216* (16 MiB). The benefits of larger blocks, and thus larger I/O, need to be weighed against the cost of COWing a giant block to modify one byte. Additionally, very large blocks can have an impact on I/O latency, and also potentially on the memory allocator. Therefore, we formerly forbade creating blocks larger

than 1M. Larger blocks could be created by changing it, and pools with larger blocks can always be imported and used, regardless of this setting.

zfs_allow_redacted_dataset_mount=**0**|1 (int)
Allow datasets received with redacted send/receive to be mounted. Normally disabled because these datasets may be missing key data.

zfs_min_metaslabs_to_flush=**1** (u64)
Minimum number of metaslabs to flush per dirty TXG.

zfs_metaslab_fragmentation_threshold=**70**% (uint)
Allow metaslabs to keep their active state as long as their fragmentation percentage is no more than this value. An active metaslab that exceeds this threshold will no longer keep its active status allowing better metaslabs to be selected.

zfs_mg_fragmentation_threshold=**95**% (uint)
Metaslab groups are considered eligible for allocations if their fragmentation metric (measured as a percentage) is less than or equal to this value. If a metaslab group exceeds this threshold then it will be skipped unless all metaslab groups within the metaslab class have also crossed this threshold.

zfs_mg_noalloc_threshold=**0**% (uint)
Defines a threshold at which metaslab groups should be eligible for allocations. The value is expressed as a percentage of free space beyond which a metaslab group is always eligible for allocations. If a metaslab group’s free space is less than or equal to the threshold, the allocator will avoid allocating to that group unless all groups in the pool have reached the threshold. Once all groups have reached the threshold, all groups are allowed to accept allocations. The default value of **0** disables the feature and causes all metaslab groups to be eligible for allocations.

This parameter allows one to deal with pools having heavily imbalanced vdevs such as would be the case when a new vdev has been added. Setting the threshold to a non-zero percentage will stop allocations from being made to vdevs that aren’t filled to the specified percentage and allow lesser filled vdevs to acquire more allocations than they otherwise would under the old **zfs_mg_alloc_failures** facility.

zfs_ddt_data_is_special=**1**|0 (int)
If enabled, ZFS will place DDT data into the special allocation class.

zfs_user_indirect_is_special=**1**|0 (int)

If enabled, ZFS will place user data indirect blocks into the special allocation class.

zfs_multihost_history=**0** (uint)
Historical statistics for this many latest multihost updates will be available in */proc/spl/kstat/zfs/<pool>/multihost*.

zfs_multihost_interval=**1000**ms (1 s) (u64)
Used to control the frequency of multihost writes which are performed when the **multihost** pool property is on. This is one of the factors used to determine the length of the activity check during import.

The multihost write period is **zfs_multihost_interval** / **leaf-vdevs**. On average a multihost write will be issued for each leaf vdev every **zfs_multihost_interval** milliseconds. In practice, the observed period can vary with the I/O load and this observed value is the delay which is stored in the uberblock.

zfs_multihost_import_intervals=**20** (uint)
Used to control the duration of the activity test on import. Smaller values of **zfs_multihost_import_intervals** will reduce the import time but increase the risk of failing to detect an active pool. The total activity check time is never allowed to drop below one second.

On import the activity check waits a minimum amount of time determined by **zfs_multihost_interval** x **zfs_multihost_import_intervals**, or the same product computed on the host which last had the pool imported, whichever is greater. The activity check time may be further extended if the value of MMP delay found in the best uberblock indicates actual multihost updates happened at longer intervals than **zfs_multihost_interval**. A minimum of *100 ms* is enforced.

0 is equivalent to **1**.

zfs_multihost_fail_intervals=**10** (uint)
Controls the behavior of the pool when multihost write failures or delays are detected.

When **0**, multihost write failures or delays are ignored. The failures will still be reported to the ZED which depending on its configuration may take action such as suspending the pool or offlining a device.

Otherwise, the pool will be suspended if **zfs_multihost_fail_intervals** x **zfs_multihost_interval** milliseconds pass without a successful MMP write. This guarantees the activity test will see MMP writes if the pool is imported. **1** is equivalent to **2**; this is necessary to prevent the pool

from being suspended due to normal, small I/O latency variations.

zfs_no_scrub_io=**0**|1 (int)
Set to disable scrub I/O. This results in scrubs not actually scrubbing data and simply doing a metadata crawl of the pool instead.

zfs_no_scrub_prefetch=**0**|1 (int)
Set to disable block prefetching for scrubs.

zfs_nocacheflush=**0**|1 (int)
Disable cache flush operations on disks when writing. Setting this will cause pool corruption on power loss if a volatile out-of-order write cache is enabled.

zfs_nopwrite_enabled=**1**|0 (int)
Allow no-operation writes. The occurrence of nopwrites will further depend on other pool properties (i.a. the checksumming and compression algorithms).

zfs_dmu_offset_next_sync=**1**|0 (int)
Enable forcing TXG sync to find holes. When enabled forces ZFS to sync data when **SEEK_HOLE** or **SEEK_DATA** flags are used allowing holes in a file to be accurately reported. When disabled holes will not be reported in recently dirtied files.

zfs_pd_bytes_max=**52428800**B (50 MiB) (int)
The number of bytes which should be prefetched during a pool traversal, like **zfs send** or other data crawling operations.

zfs_traverse_indirect_prefetch_limit=**32** (uint)
The number of blocks pointed by indirect (non-L0) block which should be prefetched during a pool traversal, like **zfs send** or other data crawling operations.

zfs_per_txg_dirty_frees_percent=**30**% (u64)
Control percentage of dirtied indirect blocks from frees allowed into one TXG. After this threshold is crossed, additional frees will wait until the next TXG. **0** disables this throttle.

zfs_prefetch_disable=**0**|1 (int)
Disable predictive prefetch. Note that it leaves "prescient" prefetch (for, e.g., **zfs send**) intact. Unlike predictive prefetch, prescient prefetch never issues I/O that ends up not being needed, so it can’t hurt performance.

zfs_qat_checksum_disable=**0**|1 (int)

Disable QAT hardware acceleration for SHA256 checksums. May be unset after the ZFS modules have been loaded to initialize the QAT hardware as long as support is compiled in and the QAT driver is present.

zfs_qat_compress_disable=**0**|1 (int)
Disable QAT hardware acceleration for gzip compression. May be unset after the ZFS modules have been loaded to initialize the QAT hardware as long as support is compiled in and the QAT driver is present.

zfs_qat_encrypt_disable=**0**|1 (int)
Disable QAT hardware acceleration for AES-GCM encryption. May be unset after the ZFS modules have been loaded to initialize the QAT hardware as long as support is compiled in and the QAT driver is present.

zfs_vnops_read_chunk_size=**1048576**B (1 MiB) (u64)
Bytes to read per chunk.

zfs_read_history=**0** (uint)
Historical statistics for this many latest reads will be available in */proc/spl/kstat/zfs/<pool>/reads*.

zfs_read_history_hits=**0**|1 (int)
Include cache hits in read history

zfs_rebuild_max_segment=**1048576**B (1 MiB) (u64)
Maximum read segment size to issue when sequentially resilvering a top-level vdev.

zfs_rebuild_scrub_enabled=**1**|0 (int)
Automatically start a pool scrub when the last active sequential resilver completes in order to verify the checksums of all blocks which have been resilvered. This is enabled by default and strongly recommended.

zfs_rebuild_vdev_limit=**67108864**B (64 MiB) (u64)
Maximum amount of I/O that can be concurrently issued for a sequential resilver per leaf device, given in bytes.

zfs_reconstruct_indirect_combinations_max=**4096** (int)
If an indirect split block contains more than this many possible unique combinations when being reconstructed, consider it too computationally expensive to check them all. Instead, try at most this many randomly selected combinations each time the block is accessed. This allows

all segment copies to participate fairly in the reconstruction when all combinations cannot be checked and prevents repeated use of one bad copy.

zfs_recover=**0**|1 (int)
Set to attempt to recover from fatal errors. This should only be used as a last resort, as it typically results in leaked space, or worse.

zfs_removal_ignore_errors=**0**|1 (int)
Ignore hard I/O errors during device removal. When set, if a device encounters a hard I/O error during the removal process the removal will not be cancelled. This can result in a normally recoverable block becoming permanently damaged and is hence not recommended. This should only be used as a last resort when the pool cannot be returned to a healthy state prior to removing the device.

zfs_removal_suspend_progress=**0**|1 (uint)
This is used by the test suite so that it can ensure that certain actions happen while in the middle of a removal.

zfs_remove_max_segment=**16777216**B (16 MiB) (uint)
The largest contiguous segment that we will attempt to allocate when removing a device. If there is a performance problem with attempting to allocate large blocks, consider decreasing this. The default value is also the maximum.

zfs_resilver_disable_defer=**0**|1 (int)
Ignore the **resilver_defer** feature, causing an operation that would start a resilver to immediately restart the one in progress.

zfs_resilver_min_time_ms=**3000**ms (3 s) (uint)
Resilvers are processed by the sync thread. While resilvering, it will spend at least this much time working on a resilver between TXG flushes.

zfs_scan_ignore_errors=**0**|1 (int)
If set, remove the DTL (dirty time list) upon completion of a pool scan (scrub), even if there were unrepairable errors. Intended to be used during pool repair or recovery to stop resilvering when the pool is next imported.

zfs_scrub_min_time_ms=**1000**ms (1 s) (uint)
Scrubs are processed by the sync thread. While scrubbing, it will spend at least this much time working on a scrub between TXG flushes.

zfs_scrub_error_blocks_per_txg=**4096** (uint)

Error blocks to be scrubbed in one txg.

zfs_scan_checkpoint_intval=**7200**s (2 hour) (uint)

To preserve progress across reboots, the sequential scan algorithm periodically needs to stop metadata scanning and issue all the verification I/O to disk. The frequency of this flushing is determined by this tunable.

zfs_scan_fill_weight=**3** (uint)

This tunable affects how scrub and resilver I/O segments are ordered. A higher number indicates that we care more about how filled in a segment is, while a lower number indicates we care more about the size of the extent without considering the gaps within a segment. This value is only tunable upon module insertion. Changing the value afterwards will have no effect on scrub or resilver performance.

zfs_scan_issue_strategy=**0** (uint)

Determines the order that data will be verified while scrubbing or resilvering:

1

Data will be verified as sequentially as possible, given the amount of memory reserved for scrubbing (see **zfs_scan_mem_lim_fact**). This may improve scrub performance if the pool’s data is very fragmented.

2

The largest mostly-contiguous chunk of found data will be verified first. By deferring scrubbing of small segments, we may later find adjacent data to coalesce and increase the segment size.

0

Use strategy **1** during normal verification and strategy **2** while taking a checkpoint.

zfs_scan_legacy=**0**|1 (int)

If unset, indicates that scrubs and resilvers will gather metadata in memory before issuing sequential I/O. Otherwise indicates that the legacy algorithm will be used, where I/O is initiated as soon as it is discovered. Unsetting will not affect scrubs or resilvers that are already in progress.

zfs_scan_max_ext_gap=**2097152**B (2 MiB) (int)

Sets the largest gap in bytes between scrub/resilver I/O operations that will still be considered sequential for sorting purposes. Changing this value will not affect scrubs or resilvers that are already in progress.

zfs_scan_mem_lim_fact=**20**^-1 (uint)

Maximum fraction of RAM used for I/O sorting by sequential scan algorithm. This tunable determines the hard limit for I/O sorting memory usage. When the hard limit is reached we stop scanning metadata and start issuing data verification I/O. This is done until we get below the soft limit.

zfs_scan_mem_lim_soft_fact=**20**^-1 (uint)

The fraction of the hard limit used to determined the soft limit for I/O sorting by the sequential scan algorithm. When we cross this limit from below no action is taken. When we cross this limit from above it is because we are issuing verification I/O. In this case (unless the metadata scan is done) we stop issuing verification I/O and start scanning metadata again until we get to the hard limit.

zfs_scan_report_txgs=**0**|1 (uint)

When reporting resilver throughput and estimated completion time use the performance observed over roughly the last **zfs_scan_report_txgs** TXGs. When set to zero performance is calculated over the time between checkpoints.

zfs_scan_strict_mem_lim=**0**|1 (int)

Enforce tight memory limits on pool scans when a sequential scan is in progress. When disabled, the memory limit may be exceeded by fast disks.

zfs_scan_suspend_progress=**0**|1 (int)

Freezes a scrub/resilver in progress without actually pausing it. Intended for testing/debugging.

zfs_scan_vdev_limit=**16777216**B (16 MiB) (int)

Maximum amount of data that can be concurrently issued at once for scrubs and resilvers per leaf device, given in bytes.

zfs_send_corrupt_data=**0**|1 (int)

Allow sending of corrupt data (ignore read/checksum errors when sending).

zfs_send_unmodified_spill_blocks=**1**|0 (int)

Include unmodified spill blocks in the send stream. Under certain circumstances, previous versions of ZFS could incorrectly remove the spill block from an existing object. Including unmodified copies of the spill blocks creates a backwards-compatible stream which will recreate a spill block if it was incorrectly removed.

zfs_send_no_prefetch_queue_ff=**20**^-1 (uint)

The fill fraction of the **zfs send** internal queues. The fill fraction controls the timing with which internal threads are woken up.

zfs_send_no_prefetch_queue_length=**1048576**B (1 MiB) (uint)

The maximum number of bytes allowed in **zfs send**'s internal queues.

zfs_send_queue_ff=**20**^-1 (uint)

The fill fraction of the **zfs send** prefetch queue. The fill fraction controls the timing with which internal threads are woken up.

zfs_send_queue_length=**16777216**B (16 MiB) (uint)

The maximum number of bytes allowed that will be prefetched by **zfs send**. This value must be at least twice the maximum block size in use.

zfs_recv_queue_ff=**20**^-1 (uint)

The fill fraction of the **zfs receive** queue. The fill fraction controls the timing with which internal threads are woken up.

zfs_recv_queue_length=**16777216**B (16 MiB) (uint)

The maximum number of bytes allowed in the **zfs receive** queue. This value must be at least twice the maximum block size in use.

zfs_recv_write_batch_size=**1048576**B (1 MiB) (uint)

The maximum amount of data, in bytes, that **zfs receive** will write in one DMU transaction. This is the uncompressed size, even when receiving a compressed send stream. This setting will not reduce the write size below a single block. Capped at a maximum of **32 MiB**.

zfs_recv_best_effort_corrective=**0** (int)

When this variable is set to non-zero a corrective receive:

1. Does not enforce the restriction of source & destination snapshot GUIDs matching.
2. If there is an error during healing, the healing receive is not terminated instead it moves on to the next record.

zfs_override_estimate_recordsize=**0**|1 (uint)

Setting this variable overrides the default logic for estimating block sizes when doing a **zfs send**. The default heuristic is that the average block size will be the current recordsize. Override this value if most data in your dataset is not of that size and you require accurate zfs send size estimates.

zfs_sync_pass_deferred_free=**2** (uint)

Flushing of data to disk is done in passes. Defer frees starting in this pass.

zfs_spa_discard_memory_limit=**16777216**B (16 MiB) (int)

Maximum memory used for prefetching a checkpoint's space map on each vdev while discarding the checkpoint.

zfs_special_class_metadata_reserve_pct=**25**% (uint)

Only allow small data blocks to be allocated on the special and dedup vdev types when the available free space percentage on these vdevs exceeds this value. This ensures reserved space is available for pool metadata as the special vdevs approach capacity.

zfs_sync_pass_dont_compress=**8** (uint)

Starting in this sync pass, disable compression (including of metadata). With the default setting, in practice, we don't have this many sync passes, so this has no effect.

The original intent was that disabling compression would help the sync passes to converge. However, in practice, disabling compression increases the average number of sync passes; because when we turn compression off, many blocks' size will change, and thus we have to re-allocate (not overwrite) them. It also increases the number of *128 KiB* allocations (e.g. for indirect blocks and spacemaps) because these will not be compressed. The *128 KiB* allocations are especially detrimental to performance on highly fragmented systems, which may have very few free segments of this size, and may need to load new metaslabs to satisfy these allocations.

zfs_sync_pass_rewrite=**2** (uint)

Rewrite new block pointers starting in this pass.

zfs_sync_taskq_batch_pct=**75**% (int)

This controls the number of threads used by **dp_sync_taskq**. The default value of **75%** will create a maximum of one thread per CPU.

zfs_trim_extent_bytes_max=**134217728**B (128 MiB) (uint)

Maximum size of TRIM command. Larger ranges will be split into chunks no larger than this value before issuing.

zfs_trim_extent_bytes_min=**32768**B (32 KiB) (uint)

Minimum size of TRIM commands. TRIM ranges smaller than this will be skipped, unless they're part of a larger range which was chunked. This is done because it's common for these small TRIMs to negatively impact overall performance.

zfs_trim_metaslab_skip=**0**|1 (uint)

Skip uninitialized metaslabs during the TRIM process. This option is useful for pools constructed from large thinly-provisioned devices where TRIM operations are slow. As a pool ages, an increasing fraction of the pool's metaslabs will be initialized, progressively degrading

the usefulness of this option. This setting is stored when starting a manual TRIM and will persist for the duration of the requested TRIM.

zfs_trim_queue_limit=**10** (uint)
Maximum number of queued TRIMs outstanding per leaf vdev. The number of concurrent TRIM commands issued to the device is controlled by **zfs_vdev_trim_min_active** and **zfs_vdev_trim_max_active**.

zfs_trim_txg_batch=**32** (uint)
The number of transaction groups' worth of frees which should be aggregated before TRIM operations are issued to the device. This setting represents a trade-off between issuing larger, more efficient TRIM operations and the delay before the recently trimmed space is available for use by the device.

Increasing this value will allow frees to be aggregated for a longer time. This will result is larger TRIM operations and potentially increased memory usage. Decreasing this value will have the opposite effect. The default of **32** was determined to be a reasonable compromise.

zfs_txg_history=**0** (uint)
Historical statistics for this many latest TXGs will be available in */proc/spl/kstat/zfs/<pool>/TXGs*.

zfs_txg_timeout=**5**s (uint)
Flush dirty data to disk at least every this many seconds (maximum TXG duration).

zfs_vdev_aggregation_limit=**1048576**B (1 MiB) (uint)
Max vdev I/O aggregation size.

zfs_vdev_aggregation_limit_non_rotating=**131072**B (128 KiB) (uint)
Max vdev I/O aggregation size for non-rotating media.

zfs_vdev_mirror_rotating_inc=**0** (int)
A number by which the balancing algorithm increments the load calculation for the purpose of selecting the least busy mirror member when an I/O operation immediately follows its predecessor on rotational vdevs for the purpose of making decisions based on load.

zfs_vdev_mirror_rotating_seek_inc=**5** (int)
A number by which the balancing algorithm increments the load calculation for the purpose of selecting the least busy mirror member when an I/O operation lacks locality as defined by **zfs_vdev_mirror_rotating_seek_offset**. Operations within this that are not immediately

following the previous operation are incremented by half.

zfs_vdev_mirror_rotating_seek_offset=**1048576**B (1 MiB) (int)
The maximum distance for the last queued I/O operation in which the balancing algorithm considers an operation to have locality. See *ZFS I/O SCHEDULER*.

zfs_vdev_mirror_non_rotating_inc=**0** (int)
A number by which the balancing algorithm increments the load calculation for the purpose of selecting the least busy mirror member on non-rotational vdevs when I/O operations do not immediately follow one another.

zfs_vdev_mirror_non_rotating_seek_inc=**1** (int)
A number by which the balancing algorithm increments the load calculation for the purpose of selecting the least busy mirror member when an I/O operation lacks locality as defined by the **zfs_vdev_mirror_rotating_seek_offset**. Operations within this that are not immediately following the previous operation are incremented by half.

zfs_vdev_read_gap_limit=**32768**B (32 KiB) (uint)
Aggregate read I/O operations if the on-disk gap between them is within this threshold.

zfs_vdev_write_gap_limit=**4096**B (4 KiB) (uint)
Aggregate write I/O operations if the on-disk gap between them is within this threshold.

zfs_vdev_raidz_impl=**fastest** (string)
Select the raidz parity implementation to use.

Variants that don't depend on CPU-specific features may be selected on module load, as they are supported on all systems. The remaining options may only be set after the module is loaded, as they are available only if the implementations are compiled in and supported on the running system.

Once the module is loaded, */sys/module/zfs/parameters/zfs_vdev_raidz_impl* will show the available options, with the currently selected one enclosed in square brackets.

fastest	selected by built-in benchmark
original	original implementation
scalar	scalar implementation

sse2	SSE2 instruction set	64-bit x86
ssse3	SSSE3 instruction set	64-bit x86
avx2	AVX2 instruction set	64-bit x86
avx512f	AVX512F instruction set	64-bit x86
avx512bw	AVX512F & AVX512BW instruction sets	64-bit x86
aarch64_neon	NEON	Aarch64/64-bit ARMv8
aarch64_neonx2	NEON with more unrolling	Aarch64/64-bit ARMv8
powerpc_altivec	Altivec	PowerPC

zfs_vdev_scheduler (charp)

DEPRECATED. Prints warning to kernel log for compatibility.

zfs_zevent_len_max=**512** (uint)

Max event queue length. Events in the queue can be viewed with zpool-events(8).

zfs_zevent_retain_max=**2000** (int)

Maximum recent zevent records to retain for duplicate checking. Setting this to **0** disables duplicate detection.

zfs_zevent_retain_expire_secs=**900**s (15 min) (int)

Lifespan for a recent ereport that was retained for duplicate checking.

zfs_zil_clean_taskq_maxalloc=**1048576** (int)

The maximum number of taskq entries that are allowed to be cached. When this limit is exceeded transaction records (itxs) will be cleaned synchronously.

zfs_zil_clean_taskq_minalloc=**1024** (int)

The number of taskq entries that are pre-populated when the taskq is first created and are immediately available for use.

zfs_zil_clean_taskq_nthr_pct=**100**% (int)

This controls the number of threads used by **dp_zil_clean_taskq**. The default value of **100%** will create a maximum of one thread per cpu.

zil_maxblocksize=**131072**B (128 KiB) (uint)

This sets the maximum block size used by the ZIL. On very fragmented pools, lowering this (typically to **36 KiB**) can improve performance.

zil_maxcopied=**7680**B (7.5 KiB) (uint)

This sets the maximum number of write bytes logged via WR_COPIED. It tunes a tradeoff between additional memory copy and possibly worse log space efficiency vs additional range lock/unlock.

zil_nocacheflush=**0**|1 (int)

Disable the cache flush commands that are normally sent to disk by the ZIL after an LWB write has completed. Setting this will cause ZIL corruption on power loss if a volatile out-of-order write cache is enabled.

zil_replay_disable=**0**|1 (int)

Disable intent logging replay. Can be disabled for recovery from corrupted ZIL.

zil_slog_bulk=**67108864**B (64 MiB) (u64)

Limit SLOG write size per commit executed with synchronous priority. Any writes above that will be executed with lower (asynchronous) priority to limit potential SLOG device abuse by single active ZIL writer.

zfs_zil_saxattr=**1**|0 (int)

Setting this tunable to zero disables ZIL logging of new **xattr=sa** records if the **org.openzfs:zilsaxattr** feature is enabled on the pool. This would only be necessary to work around bugs in the ZIL logging or replay code for this record type. The tunable has no effect if the feature is disabled.

zfs_embedded_slog_min_ms=**64** (uint)

Usually, one metaslab from each normal-class vdev is dedicated for use by the ZIL to log synchronous writes. However, if there are fewer than **zfs_embedded_slog_min_ms** metaslabs in the vdev, this functionality is disabled. This ensures that we don’t set aside an unreasonable amount of space for the ZIL.

zstd_earlyabort_pass=**1** (uint)

Whether heuristic for detection of incompressible data with zstd levels >= 3 using LZ4 and zstd-1 passes is enabled.

zstd_abort_size=**131072** (uint)

Minimal uncompressed size (inclusive) of a record before the early abort heuristic will be

attempted.

zio_deadman_log_all=**0**|1 (int)

If non-zero, the zio deadman will produce debugging messages (see **zfs_dbgmsg_enable**) for all zios, rather than only for leaf zios possessing a vdev. This is meant to be used by developers to gain diagnostic information for hang conditions which don't involve a mutex or other locking primitive: typically conditions in which a thread in the zio pipeline is looping indefinitely.

zio_slow_io_ms=**30000**ms (30 s) (int)

When an I/O operation takes more than this much time to complete, it's marked as slow. Each slow operation causes a delay zevent. Slow I/O counters can be seen with **zpool status -s**.

zio_dva_throttle_enabled=**1**|0 (int)

Throttle block allocations in the I/O pipeline. This allows for dynamic allocation distribution when devices are imbalanced. When enabled, the maximum number of pending allocations per top-level vdev is limited by **zfs_vdev_queue_depth_pct**.

zfs_xattr_compat=0|1 (int)

Control the naming scheme used when setting new xattrs in the user namespace. If **0** (the default on Linux), user namespace xattr names are prefixed with the namespace, to be backwards compatible with previous versions of ZFS on Linux. If **1** (the default on FreeBSD), user namespace xattr names are not prefixed, to be backwards compatible with previous versions of ZFS on illumos and FreeBSD.

Either naming scheme can be read on this and future versions of ZFS, regardless of this tunable, but legacy ZFS on illumos or FreeBSD are unable to read user namespace xattrs written in the Linux format, and legacy versions of ZFS on Linux are unable to read user namespace xattrs written in the legacy ZFS format.

An existing xattr with the alternate naming scheme is removed when overwriting the xattr so as to not accumulate duplicates.

zio_requeue_io_start_cut_in_line=**0**|1 (int)

Prioritize requeued I/O.

zio_taskq_batch_pct=**80**% (uint)

Percentage of online CPUs which will run a worker thread for I/O. These workers are responsible for I/O work such as compression, encryption, checksum and parity calculations. Fractional number of CPUs will be rounded down.

The default value of **80%** was chosen to avoid using all CPUs which can result in latency issues and inconsistent application performance, especially when slower compression and/or checksumming is enabled. Set value only applies to pools imported/created after that.

zio_taskq_batch_tpq=0 (uint)

Number of worker threads per taskq. Lower values improve I/O ordering and CPU utilization, while higher reduces lock contention.

If **0**, generate a system-dependent value close to 6 threads per taskq. Set value only applies to pools imported/created after that.

zio_taskq_read=fixed,1,8 null scale null (charp)

Set the queue and thread configuration for the IO read queues. This is an advanced debugging parameter. Don’t change this unless you understand what it does. Set values only apply to pools imported/created after that.

zio_taskq_write=batch fixed,1,5 scale fixed,1,5 (charp)

Set the queue and thread configuration for the IO write queues. This is an advanced debugging parameter. Don’t change this unless you understand what it does. Set values only apply to pools imported/created after that.

zvol_inhibit_dev=0|1 (uint)

Do not create zvol device nodes. This may slightly improve startup time on systems with a very large number of zvols.

zvol_major=230 (uint)

Major number for zvol block devices.

zvol_max_discard_blocks=16384 (long)

Discard (TRIM) operations done on zvols will be done in batches of this many blocks, where block size is determined by the **volblocksize** property of a zvol.

zvol_prefetch_bytes=131072B (128 KiB) (uint)

When adding a zvol to the system, prefetch this many bytes from the start and end of the volume. Prefetching these regions of the volume is desirable, because they are likely to be accessed immediately by blkid(8) or the kernel partitioner.

zvol_request_sync=0|1 (uint)

When processing I/O requests for a zvol, submit them synchronously. This effectively limits the queue depth to *1* for each I/O submitter. When unset, requests are handled asynchronously

by a thread pool. The number of requests which can be handled concurrently is controlled by **zvol_threads**. **zvol_request_sync** is ignored when running on a kernel that supports block multiqueue (blk-mq).

zvol_num_taskqs=**0** (uint)

Number of zvol taskqs. If **0** (the default) then scaling is done internally to prefer 6 threads per taskq. This only applies on Linux.

zvol_threads=**0** (uint)

The number of system wide threads to use for processing zvol block IOs. If **0** (the default) then internally set **zvol_threads** to the number of CPUs present or 32 (whichever is greater).

zvol_blk_mq_threads=**0** (uint)

The number of threads per zvol to use for queuing IO requests. This parameter will only appear if your kernel supports blk-mq and is only read and assigned to a zvol at zvol load time. If **0** (the default) then internally set **zvol_blk_mq_threads** to the number of CPUs present.

zvol_use_blk_mq=**0**|1 (uint)

Set to **1** to use the blk-mq API for zvols. Set to **0** (the default) to use the legacy zvol APIs. This setting can give better or worse zvol performance depending on the workload. This parameter will only appear if your kernel supports blk-mq and is only read and assigned to a zvol at zvol load time.

zvol_blk_mq_blocks_per_thread=**8** (uint)

If **zvol_use_blk_mq** is enabled, then process this number of **volblocksize**-sized blocks per zvol thread. This tunable can be use to favor better performance for zvol reads (lower values) or writes (higher values). If set to **0**, then the zvol layer will process the maximum number of blocks per thread that it can. This parameter will only appear if your kernel supports blk-mq and is only applied at each zvol's load time.

zvol_blk_mq_queue_depth=**0** (uint)

The queue_depth value for the zvol blk-mq interface. This parameter will only appear if your kernel supports blk-mq and is only applied at each zvol's load time. If **0** (the default) then use the kernel's default queue depth. Values are clamped to the kernel's BLKDEV_MIN_RQ and BLKDEV_MAX_RQ/BLKDEV_DEFAULT_RQ limits.

zvol_volmode=**1** (uint)

Defines zvol block devices behaviour when **volmode**=**default**:

1

equivalent to **full**

2
equivalent to **dev**
3
equivalent to **none**

zvol_enforce_quotas=**0**|1 (uint)
Enable strict ZVOL quota enforcement. The strict quota enforcement may have a performance impact.

ZFS I/O SCHEDULER

ZFS issues I/O operations to leaf vdevs to satisfy and complete I/O operations. The scheduler determines when and in what order those operations are issued. The scheduler divides operations into five I/O classes, prioritized in the following order: sync read, sync write, async read, async write, and scrub/resilver. Each queue defines the minimum and maximum number of concurrent operations that may be issued to the device. In addition, the device has an aggregate maximum, **zfs_vdev_max_active**. Note that the sum of the per-queue minima must not exceed the aggregate maximum. If the sum of the per-queue maxima exceeds the aggregate maximum, then the number of active operations may reach **zfs_vdev_max_active**, in which case no further operations will be issued, regardless of whether all per-queue minima have been met.

For many physical devices, throughput increases with the number of concurrent operations, but latency typically suffers. Furthermore, physical devices typically have a limit at which more concurrent operations have no effect on throughput or can actually cause it to decrease.

The scheduler selects the next operation to issue by first looking for an I/O class whose minimum has not been satisfied. Once all are satisfied and the aggregate maximum has not been hit, the scheduler looks for classes whose maximum has not been satisfied. Iteration through the I/O classes is done in the order specified above. No further operations are issued if the aggregate maximum number of concurrent operations has been hit, or if there are no operations queued for an I/O class that has not hit its maximum. Every time an I/O operation is queued or an operation completes, the scheduler looks for new operations to issue.

In general, smaller **max_active**s will lead to lower latency of synchronous operations. Larger **max_active**s may lead to higher overall throughput, depending on underlying storage.

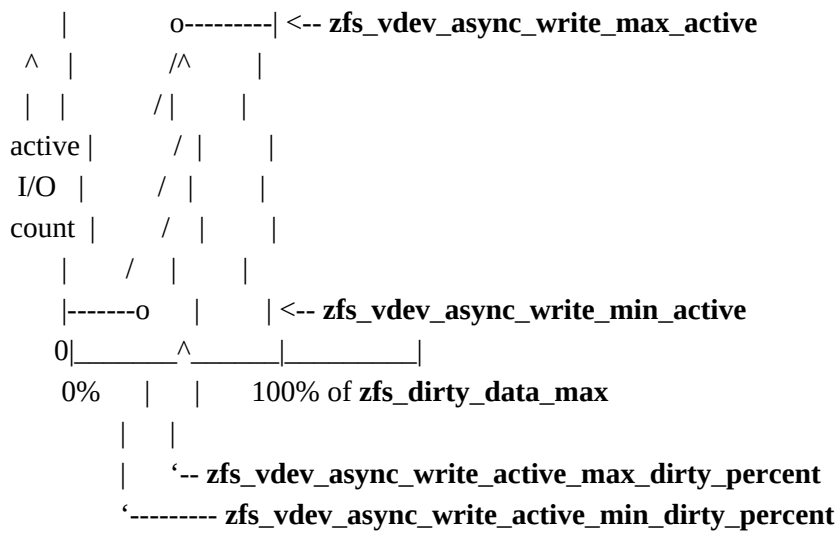
The ratio of the queues' **max_active**s determines the balance of performance between reads, writes, and scrubs. For example, increasing **zfs_vdev_scrub_max_active** will cause the scrub or resilver to complete more quickly, but reads and writes to have higher latency and lower throughput.

All I/O classes have a fixed maximum number of outstanding operations, except for the async write

class. Asynchronous writes represent the data that is committed to stable storage during the syncing stage for transaction groups. Transaction groups enter the syncing state periodically, so the number of queued async writes will quickly burst up and then bleed down to zero. Rather than servicing them as quickly as possible, the I/O scheduler changes the maximum number of active async write operations according to the amount of dirty data in the pool. Since both throughput and latency typically increase with the number of concurrent operations issued to physical devices, reducing the burstiness in the number of simultaneous operations also stabilizes the response time of operations from other queues, in particular synchronous ones. In broad strokes, the I/O scheduler will issue more concurrent operations from the async write queue as there is more dirty data in the pool.

Async Writes

The number of concurrent operations issued for the async write I/O class follows a piece-wise linear function defined by a few adjustable points:



Until the amount of dirty data exceeds a minimum percentage of the dirty data allowed in the pool, the I/O scheduler will limit the number of concurrent operations to the minimum. As that threshold is crossed, the number of concurrent operations issued increases linearly to the maximum at the specified maximum percentage of the dirty data allowed in the pool.

Ideally, the amount of dirty data on a busy pool will stay in the sloped part of the function between **zfs_vdev_async_write_active_min_dirty_percent** and **zfs_vdev_async_write_active_max_dirty_percent**. If it exceeds the maximum percentage, this indicates that the rate of incoming data is greater than the rate that the backend storage can handle. In this case, we must further throttle incoming writes, as described in the next section.

ZFS TRANSACTION DELAY

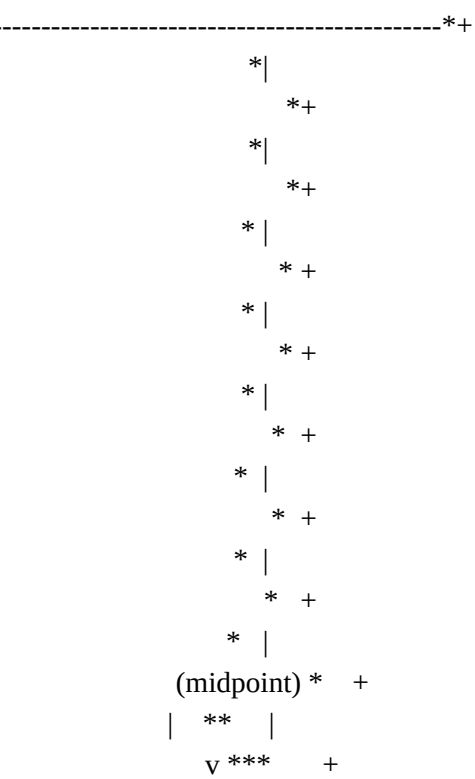
We delay transactions when we’ve determined that the backend storage isn’t able to accommodate the rate of incoming writes.

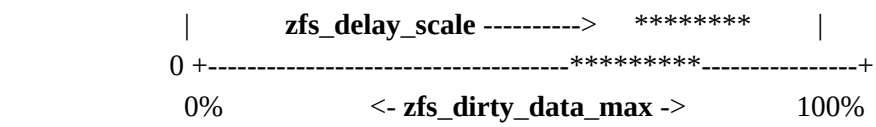
If there is already a transaction waiting, we delay relative to when that transaction will finish waiting. This way the calculated delay time is independent of the number of threads concurrently executing transactions.

If we are the only waiter, wait relative to when the transaction started, rather than the current time. This credits the transaction for "time already served", e.g. reading indirect blocks.

The minimum time for a transaction to take is calculated as
min_time = min(**zfs_delay_scale** x (**dirty** - **min**) / (**max** - **dirty**), 100ms)

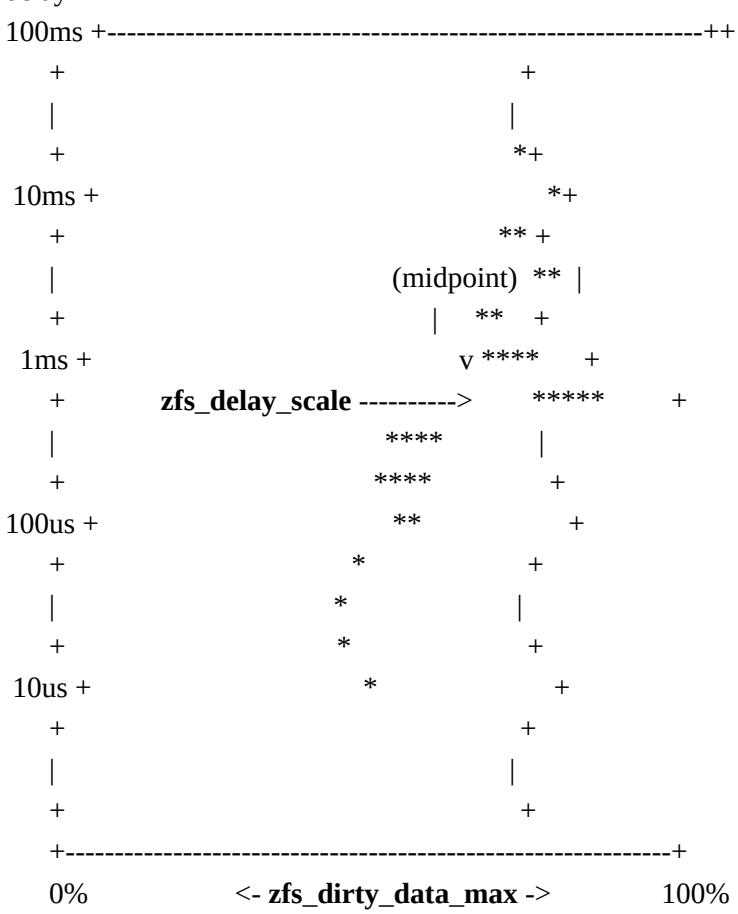
The delay has two degrees of freedom that can be adjusted via tunables. The percentage of dirty data at which we start to delay is defined by **zfs_delay_min_dirty_percent**. This should typically be at or above **zfs_vdev_async_write_active_max_dirty_percent**, so that we only start to delay after writing at full speed has failed to keep up with the incoming write rate. The scale of the curve is defined by **zfs_delay_scale**. Roughly speaking, this variable determines the amount of delay at the midpoint of the curve.





Note, that since the delay is added to the outstanding time remaining on the most recent transaction it's effectively the inverse of IOPS. Here, the midpoint of *500 us* translates to *2000 IOPS*. The shape of the curve was chosen such that small changes in the amount of accumulated dirty data in the first three quarters of the curve yield relatively small differences in the amount of delay.

The effects can be easier to understand when the amount of delay is represented on a logarithmic scale:



Note here that only as the amount of dirty data approaches its limit does the delay start to increase rapidly. The goal of a properly tuned system should be to keep the amount of dirty data out of that range by first ensuring that the appropriate limits are set for the I/O scheduler to reach optimal throughput on the back-end storage, and then by changing the value of **zfs_delay_scale** to increase the steepness of the curve.