# Why fsync() on OpenZFS can’t fail, and what happens when it does

Rob Norris, *Klara Inc.*

## Abstract

On OpenZFS, `fsync()` cannot fail - it will wait until the application’s changes are on disk before it returns. If there is a problem, such as a hardware failure, that causes the pool to suspend, then it may wait forever. This feels strange, but is acceptable according to the API contract: `fsync()` never returned success, so the application has no reason to believe its data is on disk.

However, OpenZFS pools can recover if the fault is repaired, and so `fsync()` can still return. As it turns out though, it’s possible in rare situations for the pool to return to service but not have actually put the data on disk. `fsync()` returns success, because it cannot fail, and the application has been lied to.

This paper describes the path taken from the `fsync()` call, through the ZFS Intent Log, the transaction machinery, the pool failure system and the IO pipeline to understand what happens to IO when disks fail, and why OpenZFS believed that writes had succeeded when they had not. It goes on to describe changes to make OpenZFS understand that something had gone wrong and respond appropriately, such that `fsync()` once again cannot fail.

## 1. Background

In early 2023 an organisation with a huge deployment of OpenZFS pools came to us with a problem. This organisation runs a cloud storage service, where they receive files from customers and store them to an OpenZFS dataset. To ensure that the files were stored safely, they would call `fsync()` at the end of each file write, and not return a “success” message to their customer until `fsync()` completed successfully, guaranteeing that the file was safely and reliably stored.

They were having problems with their SAS backplanes. At times, for no apparent reason, they would lose power, appearing to the OS that multiple disks had failed at the same moment. When this occurred OpenZFS would simply wait. After a short time, the backplane would return, and OpenZFS would continue. This was not ideal, but was determined to be a fault in the backplane itself.

Sometimes, after this happened, an attempt to read a file would result in an IO error. Investigation found that the `fsync()` call was in progress at the time the backplane failed, and after it returned and OpenZFS recovered, the `fsync()` call had returned success, even though the file was clearly not correctly written to disk.

We were asked to investigate what was happening and make appropriate modifications to OpenZFS to ensure that `fsync()` never returns success unless the file is really properly stored.

# 2. The OpenZFS write path

This section provides a brief overview of how filesystem modifications submitted by a user application make their way through OpenZFS and out to disk. Readers familiar with OpenZFS internals will note that this is a rather simplified view, but is all the context needed to understand the rest of this paper.

## *2.1. The DMU and the SPA*

The Data Management Unit (DMU) is a component inside OpenZFS that provides the notion of "objects": arbitrarily-sized runs of data, named by a single 64-bit identifier. The facilities used by user programs to interact with OpenZFS, like filesystems (files, directories), volumes and even whole datasets are all implemented as different kinds of objects.

Under the DMU is the Storage Pool Allocator (SPA). This is the component of OpenZFS that manages the physical devices and allocates "blocks" from them to be used for storage. Each DMU object is constructed from a tree of blocks.

## *2.2. Transactions*

The SPA and the DMU share the notion of a "transaction". Modifications made to a DMU object (eg a file) are accumulated in memory, and assigned to a transaction. After some time, the transaction is closed and starts being written out to disk, and a new one is opened for the next round of changes to be assigned to. As such, a transaction is the fundamental unit of storage atomicity in OpenZFS.

## *2.3. Transaction sync*

After a transaction is closed, the DMU iterates over all the changes it has accumulated and submits them to the SPA as writing IO requests. Because they all belong to the same transaction, they are submitted in a way that if it one causes an IO error, they all fail with that error. This is important, as there are different subsystems all over OpenZFS waiting for their return before moving on to something else, so it does no good to return success for a specific change if the entire transaction failed.

## *2.4. Labels and uberblocks*

Once the data writes are completed, the pool-wide "uberblocks" and the per-device "labels" are updated with information about the new transaction. These are what OpenZFS looks for when importing a pool, so until they're fully written out, the transaction does not "exist", as such.

These are written in a special sequence. First the "even" label is written to each device and flushed, then the uberblocks are written and flushed, then the "odd" label is written and flushed. This is done so that if power is lost, at import OpenZFS can tell where in the sequence it failed, and understand which transaction is the "correct" one.

## *2.5. Signal waiters*

Once the transaction is fully written out, waiters all over OpenZFS are woken to inform them of this fact. There's many that might be waiting depending what's going on at the time, not least the thread waiting to begin syncing the next transaction out.

# 3. How OpenZFS implements `fsync()`

## *3.1. What is `fsync()`?*

The `fsync()` system call has the prototype:

```
int fsync(int fildes);
```

The common understanding is that when called on an open file, its successful return indicates that the file is durably stored on disk. The expectation is that an unexpected system failure (eg power loss) at that moment would not affect the file, and after the system is recovered the file contains all the data written up to the `fsync()` call. It is often used as a building block for transactional IO.

## *3.2. The ZFS Intent Log*

OpenZFS' transactional model presents a challenge for implementing a system call like `fsync()`, which requested that all changes for a single file be stored immediately. Changes accumulated in a transaction may touch multiple files, and they may be dependent on each other. It's not really possible to commit the changes for a single file separately from the rest of the transaction. One could make `fsync()` forcibly commit the entire transaction, but that can be very slow if the requested file was just one of thousands modified in the transaction.

OpenZFS handles this situation as many other storage systems do, by keeping a "journal" or "intent log" of the logical changes that have been made on each file. This component inside OpenZFS is known as the ZFS Intent Log, or ZIL.

Every modification made to a file is now stored in memory twice: once in the Data Management Unit (DMU) as a normal part of the file state, and again in the ZIL as a record of the operations taken to get the file to that state.

For example, a typical file write sequence might look like:

```
int fd = open("somefile", O_CREAT | O_WRONLY);
write(fd, "somedata", 8);
write(fd, "moredata", 8);
close(fd);
```

In the ZIL, this would generate a sequence of intent log entries. Simplified, they might look like:

```
TX_CREATE   <dirobj> <fileobj> "somefile"
TX_WRITE    <fileobj> 0 8 "somedata"
TX_WRITE    <fileobj> 8 16 "moredata"
```

Then, when `fsync()` is called on a file, the relevant portions of this list are immediately written to a special "replay list" associated with each dataset. Only the entries for files specifically requested by `fsync()`, and any that were modified as a result, are actually added to the replay list. In this case, `fsync()` on the file we created would also add any modifications for the directory object to the replay list.

When the transaction completes and is written down to its final resting place, all in-memory intent log entries are destroyed, and the on-disk replay list is deleted. This means there's only minimal overhead if nothing uses `fsync()` - the intent log entries are only ever in memory.

The last piece of the ZIL is the replay facility. When a dataset is mounted, the replay list is checked. Any operations there are applied, so the filesystem state moves forward from whatever it was at the last completed transaction to also have any changes that were explicitly `fsync()`'d during the last incomplete transaction.

### *3.3. Transaction sync fallback*

If writing the ZIL replay list fails, the `fsync()` call simply falls back to waiting for the transaction to be committed. This is the situation the ZIL exists to avoid, but it is the only remaining option that preserves `fsync()`'s API guarantee. When the transaction is successfully committed, the changes are on disk, and `fsync()` can return success.

## 4. How OpenZFS handles device failure

### *4.1. Device probes*

OpenZFS takes a passive approach to detecting device failure. In the general case, it will assume that a device exists and is in good health until and unless an IO request sent to it returns a error.

Most errors will result in a "probe" sequence being sent to the device. First, an attempt is made to read from the labels stored on that device, which are always in a known location. If the read succeeds, an attempt to write a new label is made. The outcome of these tests are saved, and will decide how the device is used in the future; for example, if a device can't be written to, then no further attempts will be made to store new data to it.

Once the probe sequence has completed, the original IO request is reprocessed. If the probe failed, the IO request will be forced to have a "no such device" error (`ENXIO`) and returned to the caller. Often, an interior vdev (eg a raidz control vdev) can compensate for this, reassigning the IO to a different physical device and retrying it. This is how pool redundancy options typically work.

If the probe succeeded, then the IO request will be retried. If it still fails, then there is clearly something very strange going on, and the pool is suspended. The original IO and any others in flight are held in a suspend list. This has the effect that all user operations block. Once an operator has corrected the fault and run `zpool clear`, failed devices are re-probed and if everything looks good, the suspended IO is reissued.

This describes the most common failure process. OpenZFS has many different kinds of IO and many possible ways to set up a pool, so there's many other things that it tries to correct for, and many special cases. There are two place in particular where something different happens that are relevant to our secnario.

### *4.2. ZIL write failure*

ZIL writes are issued with a special IO request flag, `ZIO_FLAG_CANFAIL`. If this type of request fails, the device is still probed and removed from service if the probe fails, but the original request will not be retried, instead just returning the original error. The ZIL does this because a failed retry can suspend the pool, which is a rather heavy-handed response when the ZIL has a fallback available - simply waiting for the transaction to complete. If the failure is transient, then the transaction will complete normally and the data will be safely on disk, allowing `fsync()` to return success.

### *4.3. Flush failure*

Much like filesystems, hard drives typically have their own internal caches that they use to improve performance. As such, a successful response to a disk write IO may not mean that the data has been physically written to the disk yet, and if power to the device is lost, so too may the data be.

To help with this, hard drives offer a "flush" operation that forces cached data to be written out, not unlike what `fsync()` is to filesystems. As with filesystems, there are several flushing operations available. OpenZFS makes use of a "write barrier" flushing operation, which is sent independent of other IO types. Its purpose is to completely write everything in the drive cache to the physical disk. OpenZFS issues device flushes to all devices in the pool after uberblock, label and ZIL writes complete successfully.

However, and somewhat surprisingly, OpenZFS ignores the response to flush requests, and has done so since ZFS was first integrated into Solaris.[1] The reasons for this are unclear, but this author's guess is that initially hard drives generally did not have cache memories or flushing operations, and even once they became widespread an uncaught flush error was extremely unlikely to not be accompanied by related write error or some other significant hardware fault, so any problem that has been seen was likely discounted as a freak occurrence.

## 5. How `fsync()` fails

### *5.1. Guaranteed success*

In OpenZFS, `fsync()` only ever returns `0`, indicating success.

Internally, `fsync()` is implemented by `zil_commit()`, which presents an API contract that guarantees success:

```
void zil_commit(zilog_t *zilog, uint64_t oid);
```

To make this guarantee, it relies on the fact that if something goes wrong in its own IO, it can fall back to waiting for the whole transaction to be committed:

```
void txg_wait_synced(struct dsl_pool *dp, uint64_t txg);
```

Given these API contracts, `fsync()` could not be any other way.

This guarantee is a reflection of OpenZFS' fundamental goal of never losing data. The systems described in the previous sections all work together to ensure that OpenZFS is always aware of where in the pipeline data is, and it is fundamentally designed to stop the world by suspending the pool if it cannot find a way to make progress. By design, there is no path through the system whereby a write operation could fail without the entire system failing, so it is sufficient to assume that forward progress means everything is in good shape.

Unfortunately, because flush errors are not checked, this only holds if writes are guaranteed to succeed. Ironically, the very presence of a flush operation makes it clear that this guarantee cannot possibly be true in all cases.

## *5.2. Flushed away*

This problem was first presented to us with a failed backplane, but it can also apply to failures in disks, cables, disk units or power supplies. If the connection is lost to the disk, the OS notices and takes the disk offline, and any further IO receives an immediate error. If the error is in response to a write request, OpenZFS notices and responds as described above. If the failure occurs between write and flush however, OpenZFS won't notice and will proceed, believing all is well.

When the ZIL write succeeds, `fsync()` returns success. If the connection to the disk fails just after the write and the flush fails, OpenZFS doesn't know, but even if it did, it would be too late - the user program has already moved on. This is often survivable - if the pool is still able to operate without the failed disk, the transaction can still be written out correctly, and the "lost" ZIL write is not needed.

If, on the other hand, the pool does not have sufficient redundancy (eg the whole backplane failing), the tree of writes for the entire transaction will fail shortly after. This happens during the label/uberblock flush sequence, so the pool will suspend and the failed IOs will be held. Usually, if the failure can be corrected, the pool will resume, replay the lost IO, and the data will end up being written out correctly.

What often happens in this situation is that an operator will take some service continuity action, like begin a service failover to another machine, and then power-cycle the system. After correcting the fault, the pool is imported, and recovery actions run. The ZIL replay log will be scanned, but since the flush failed, some or all of the log can be missing. The write that OpenZFS promised made it to disk (via its successful `fsync()` return) is permanently lost.

# 6. Constructing a test case

## *6.1. Writing some files*

As we investigated the problem we built a test case to demonstrate it. The organisation that first saw the problem already had a program that simulated their workload, which we adapted to help us understand the problem.

The program creates thousands of directories, and then creates 100 threads that run a tight loop, creating a new file in a random directory, writing a block of random data of length 4K-1.5M, renaming it into place, then calling `fsync()` to force it to disk, closing it, and moving on to the next. Multiple directories are used to avoid threads contending on directory updates.

Each step in the sequence is named:

- `open`: create the file
- `stat`: get its inode number, that is, OpenZFS object id
- `header`: write the header, a text representation of the number of bytes we're about to write
- `write`: write the actual data, some amount of random data
- `footer`: write a fixed text footer
- `rename`: rename the file to its final name
- `sync`: call `fsync()`
- `done`: close the file, success!

For every file, we log its result, either success:

```
[pool-655033360322456518] 01:04:31.143215 SUCCESS: path=/pool-655033360322456518/dir-
14/13.tmp; ino=16386; size=447584
```

or if it fails, details about how far into that sequence it got:

```
[pool-655033360322456518] 06:31:15.392190 FAILED: path=/pool-655033360322456518/dir-
10074/58073.tmp.new; ino=0; size=812014; step=open; err=5
```

This allows us to understand what operations the program took, and the result, which we can then compare with what we see on disk and understand if OpenZFS did the right thing.

## *6.2. Power off*

We let this program run for a couple of minutes, and then simulate a total pool failure by instructing the SAS backplane to power off a number of disks. These tests are run on 14-wide RAIDz3 pools; to ensure failure we power off five disks. The pool promptly suspends:

```
  pool: pool-655033360322456518
 state: SUSPENDED
status: One or more devices are faulted in response to IO failures.
action: Make sure the affected devices are connected, then run 'zpool clear'.
   see: https://openzfs.github.io/openzfs-docs/msg/ZFS-8000-HC
config:
        NAME                        STATE     READ WRITE CKSUM
        pool-655033360322456518     ONLINE       0     0     0
          raidz3-0                  ONLINE       0     2     0
            wwn-0x5000cca26438c380  ONLINE       3     4     0
            wwn-0x5000cca2647dd640  ONLINE       3     4     0
            wwn-0x5000cca264800930  ONLINE       3     7     0
            wwn-0x5000cca2648defa8  ONLINE       3     5     0
            wwn-0x5000cca2648ee8d8  ONLINE       3    11     0
            wwn-0x5000cca2649004cc  ONLINE       0     0     0
            wwn-0x5000cca2649042e8  ONLINE       0     0     0
            wwn-0x5000cca26490a5fc  ONLINE       0     0     0
            wwn-0x5000cca2649116c0  ONLINE       0     0     0
            wwn-0x5000cca26491175c  ONLINE       0     0     0
            wwn-0x5000cca26491428c  ONLINE       0     0     0
            wwn-0x5000cca2649147c4  ONLINE       0     0     0
            wwn-0x5000cca2649173e4  ONLINE       0     0     0
            wwn-0x5000cca26491b4b4  ONLINE       0     0     0
```

At this point, OpenZFS is holding all the failed IO, waiting for the pool to return. If we powered the disks on right now, it would almost certainly recover. To ensure that everything is lost and we’re left with only what’s already on the disks, so we instruct the BMC to hard power off the machine.

## *6.3. Log analysis*

Once the machine returns, we power back on the disks and import the pool. Then we run a program that loads and parses the result log, and then reads every file on the pool to see if it matches what we think we wrote. The output looks something like:

```
      100  BROKEN: reported success, but found problem: file not on disk
    46185  OK: wrote correctly, found on disk as expected

BROKEN FILES (damaged on disk, but logged success):
  ts=06:46:01.612511 ino=64527 path=/pool-12255040345250044327/dir-14189/46188.tmp
problem=file not on disk
  ts=06:46:01.612549 ino=63881 path=/pool-12255040345250044327/dir-14187/46186.tmp
problem=file not on disk
  ts=06:46:01.612593 ino=64144 path=/pool-12255040345250044327/dir-14213/46212.tmp
problem=file not on disk
...
```

According to the result log, all of these were ones that completed their entire sequence and `fsync()` returned success for, eg:

```
[pool-12255040345250044327] 06:46:01.612511 SUCCESS: path=/pool-12255040345250044327dir-
14189/46188.tmp; ino=64527; size=465053
```

Because this problem is sensitive to timing, it’s possible to complete a test run and for everything to be fine. In our experience that was rare. To be sure, when we felt we had a good fix, we’d do multiple runs to confirm it.

# 7. Handling ZIL flush failures

## *7.1. Propagate flush*

The flushing function `zio_flush()` is very simple:

```
void
zio_flush(zio_t *zio, vdev_t *vd)
{
        zio_nowait(zio_ioctl(zio, zio->io_spa, vd, DKIOCFLUSHWRITECACHE,
            NULL, NULL,
            ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_PROPAGATE | ZIO_FLAG_DONT_RETRY));
}
```

The flag combination is about the most aggressive way to issue IO and not care what happens:

- `DONT_RETRY`: if this operation fails, don't bother trying again
- `CANFAIL`: if this operation fails, don't suspend the pool
- `DONT_PROPAGATE`: if this operation fails, don't tell me about it

`DONT_RETRY` and `CANFAIL` are probably reasonable; the success or failure of a flush is a crucial piece of information we need to manage the pool state. The IO layer shouldn't blindly retry on error and assume all is well if the retry succeeds, but neither should it just blindly suspend the entire pool since the meaning of the error will depend on the context the request was issued in.

`DONT_PROPAGATE` however is obvious not reasonable. A flush failure means something, and we have to tell someone!

Our first attempt to fix this was to simply propagate flush errors into the ZIL. The ZIL code is already structured to support this; writes and flush IOs are already properly chained together such that a flush failure would cause the write to fail, and there are comments explaining how it would work, if only flush errors were propagated:

```
                /*
                 * We expect any ZIO errors from child ZIOs to have been
                 * propagated "up" to this specific LWB's root ZIO, in
                 * order for this error handling to work correctly. This
                 * includes ZIO errors from either this LWB's write or
                 * flush, as well as any errors from other dependent LWBs
                 * (e.g. a root LWB ZIO that might be a child of this LWB).
                 *
                 * With that said, it's important to note that LWB flush
                 * errors are not propagated up to the LWB root ZIO.
                 * This is incorrect behavior, and results in VDEV flush
                 * errors not being handled correctly here. See the
                 * comment above the call to "zio_flush" for details.
                 */
```

and:

```
                vdev_t *vd = vdev_lookup_top(spa, zv->zv_vdev);
                if (vd != NULL) {
                        /*
                         * The "ZIO_FLAG_DONT_PROPAGATE" is currently
                         * always used within "zio_flush". This means,
                         * any errors when flushing the vdev(s), will
                         * (unfortunately) not be handled correctly,
                         * since these "zio_flush" errors will not be
                         * propagated up to "zil_lwb_flush_vdevs_done".
                         */
                        zio_flush(lwb->lwb_root_zio, vd);
                }
```

So, we adjusted `zio_flush()` to allow the caller to request propagation:

```
void
zio_flush(zio_t *zio, vdev_t *vd, boolean_t propagate)
{
        zio_nowait(zio_ioctl(zio, zio->io_spa, vd, DKIOCFLUSHWRITECACHE,
            NULL, NULL, ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_RETRY |
            (propagate ? ZIO_FLAG_PROPAGATE : 0)));
}
```

This actually worked! When we ran our test case, while sometimes damage was found on the pool, we never saw a case where `fsync()` incorrectly returned success.

However, this turned out to be not as complete a fix as we hoped.

## 7.2. Too much flushing

Most IO requests in OpenZFS are issued to a top-level vdev (eg a raidz or mirror) and lets that vdev decide how to distribute that to its children. `zio_flush()` looks like that from the outside, but the function under it, `zio_ioctl()`, doesn't actually work that way.

```
zio_t *
zio_ioctl(zio_t *pio, spa_t *spa, vdev_t *vd, int cmd,
    zio_done_func_t *done, void *private, enum zio_flag flags)
{
        zio_t *zio;
        int c;

        if (vd->vdev_children == 0) {
                zio = zio_create(pio, spa, 0, NULL, NULL, 0, 0, done, private,
                    ZIO_TYPE_IOCTL, ZIO_PRIORITY_NOW, flags, vd, 0, NULL,
                    ZIO_STAGE_OPEN, ZIO_IOCTL_PIPELINE);

                zio->io_cmd = cmd;
        } else {
                zio = zio_null(pio, spa, NULL, NULL, NULL, flags);

                for (c = 0; c < vd->vdev_children; c++)
                        zio_nowait(zio_ioctl(zio, spa, vd->vdev_child[c], cmd,
                            done, private, flags));
        }

        return (zio);
}
```

Instead, it walks the vdev tree building a matching tree of IO requests, and true physical IO requests are only issued to leaf, that is, “real” devices. So in our case, all 12 drives in the pool receive a flush.

At minimum, this is wasteful, as we’re sending flushes to disks that we never wrote anything to, and waiting for them to respond. The real problem however is that the entire top-level operation succeeds if *all* leaf operations succeed, but fails if *any* leaf operations fail.

If we’ve lost enough disks to suspend the entire pool, this doesn’t matter - several disks really will fail their flush operation, and the pool will rightly suspend. However in the case of a regular degraded pool, where only a single disk is lost, that disk will also return failure to its flush operation, and again, the entire top-level flush operation fails.

For our ZIL flushes, that will also fail the ZIL write, which will cause the ZIL to fallback to waiting for the transaction to be written out, and `fsync()` to return success. This destroys performance - now *every* call to `fsync()` has to wait, as though the ZIL didn’t exist at all.

In addition to recording file results, the test program also records the average time spent in each stage. Normally, with the pool in good health, it would show an average `fsync()` time of ~120ms.

```
TIME     OPEN    WRITE      SYNC
  10    0.246    0.961   120.000
  20    0.251    0.711   119.000
  30    0.246    0.704   116.000
  40    0.248    0.668   116.000
  50    0.252    0.707   123.000
  60    0.246    0.685   122.000
```

With a single failed disk, and the change to enable flush propagation above, `fsync()` blows out:

```
  70    0.195    0.639   333.000
  80    0.187    0.604   251.000
  90    0.174    0.621   241.000
 100    0.180    0.614   237.000
 110    0.183    0.592   245.000
 120    0.182    0.620   247.000
```

## *7.3. An unknown amount of flushing*

We spent a long time experimenting with different ways of trying to work around this, with limited success, before finally understanding the problem and fixing it properly.

The original problem was that we just received a successful response to a ZIL write, but we don’t know if it’s truly on disk:

- The pool might be in perfect shape, and the write happened normally.
- The write may have failed initially, and OpenZFS probed a disk, found it failed, changed its state to “degraded” and then retried the write to different disks, which succeeded.

- The pool may be critically damaged, as in our original backplane failure, but the write has still succeeded.
- An operator may have removed a disk in a raidz vdev, and the write successfully happened to other disks in good health.

What we really need to do is issue a flush for this IO, that is, we need to tell all the disks that were written to that they need to do a flush. At the same time, we must avoid any disks that we *didn't* write to, as we don't know what they'll do. This means we need to somehow bind flushes to writes, so we only flush what was written, and no more.

The problem is that at this layer within OpenZFS, all we have is the completed IO object. We don't know how it was serviced, by design - the entire point of the SPA is that that it presents a uniform interface to all kinds of pool topologies. We have no idea what happens under the hood.

An early attempt to resolve this problem was to add an IO flag to indicate that this IO should be flushed as it's written. This is analogous to the SCSI Force Unit Access (FUA) flag, which enables a write-through (no cache) mode. This worked, in that the performance profile was not affected by a degraded device, but it still reduced `fsync()` performance, for two reasons:

- it forced a flush after every write, even when multiple writes for the same disk were on the IO tree
- it subverted a feature of the ZIL, which is to delay flushing when there's nothing actively waiting for a flushing operation (for the case where a `fsync()` also required other objects to be flushed, even though there's nothing explicitly waiting for them)

In the end, we decided that there was no other way, and we would have to know exactly which disks we had written to in order to be sure that our writes had completed.

## 7.4. Tracing IO

Since we had to hook something deeper into the IO layer to make this work, we tried to make something reasonably generic, rather than tied specifically to flushing.

We introduced the notion of "vdev tracing". When creating an IO request, a caller can add the `ZIO_FLAG_VDEV_TRACE` flag to indicate that this IO should be traced.

A traced request is executed as normal: each stage of the IO pipeline may create child IO objects, and those create child IO objects, all the way down the the real disks. As child IO objects complete, as well as propagating their error code we now also propagate a list of any vdevs that returned a successful response while executing the IO. There are no other behaviour changes. Thus, when the original request is completed, it will still have the same overall success/fail response, and if it succeeds it will also have an attached list of all vdevs that were involved in its success. This list can then be used to instruct those vdevs perform any follow-up operations, such as a flush.

By making our ZIL write request a traced request, so its completion handler can know where to issue a flush to.

## 7.5. The right amount of flushing

Armed with this new facility, we can write a new flushing function that only issues flushes to the vdevs involved in servicing a "traced" IO:

```
void
zio_flush_traced(zio_t *pio, zio_t *tio, boolean_t propagate)
```

```
{
        const int cmd = DKIOCFLUSHWRITECACHE;
        const zio_flag_t flags =
            ZIO_FLAG_CANFAIL | ZIO_FLAG_DONT_RETRY |
            (propagate ? 0 : ZIO_FLAG_DONT_PROPAGATE);
        spa_t *spa = pio->io_spa;
        zio_t *fio, *zio;
        zio_vdev_trace_t *zvt;
        vdev_t *vd;

        fio = zio_null(pio, spa, NULL, NULL, NULL, flags);

        for (zvt = avl_first(tio->io_vdev_trace_tree); zvt != NULL;
            zvt = AVL_NEXT(tio->io_vdev_trace_tree, zvt)) {
                vd = vdev_lookup_by_guid(spa->spa_root_vdev, zvt->zvt_guid);
                if (vd == NULL)
                        continue;
                if (vd->vdev_children == 0) {
                        zio = zio_create(fio, spa, 0, NULL, NULL, 0, 0, NULL,
                            NULL, ZIO_TYPE_IOCTL, ZIO_PRIORITY_NOW, flags, vd,
                            0, NULL, ZIO_STAGE_OPEN, ZIO_IOCTL_PIPELINE);
                        zio->io_cmd = cmd;
                        zio_nowait(zio);
                }
        }

        zio_nowait(fio);
}
```

And then in the ZIL write completion handler, its just a case of trading `zio_flush()` for `zio_flush_traced()`:

```
                zio_flush_traced(lwb->lwb_root_zio, zio, B_TRUE);
```

When we run the performance test again with a disk missing, we see the positive result:

```
TIME     OPEN    WRITE      SYNC
  10    0.278    0.938   121.000
  20    0.287    0.757   123.000
  30    0.284    0.695   124.000
  40    0.285    0.699   122.000
  50    0.293    0.702   134.000
  60    0.290    0.678   123.000
  70    0.287    0.656   178.000
  80    0.298    0.659   118.000
  90    0.299    0.747   124.000
 100    0.298    0.653   120.000
 110    0.291    0.692   122.000
 120    0.284    0.677   122.000
```

A single disk is powered off ~70s. This does trigger some amount of ZIL write failures and flush failures, causing some amount of fallback to transaction sync and thus a brief bump in the average sync time, but it quickly settles back to the normal average once things settle and the surviving disks pick up normal service for the pool.

# 8. Status and further work

The two major changes described here – flush error propagation and vdev tracing – are in production at the sponsoring organisation, and are working as planned.

The intent is to offer these changes to upstream OpenZFS during 2024. These changes were developed against OpenZFS 2.1.5, so need to be forward-ported to the current development branch, and properly repeatable tests developed.

More study is required to understand if it's necessary to extend flush error propagation to the transaction commit sequence of writing labels and uberblocks and if so, how to do so safely.

# Acknowledgements

This work is complex, and its study and resolution involved multiple people over many months. Thanks to the team at Klara Inc., past and and present, for their careful and thoughtful analysis and contributions. In particular, thanks to Allan Jude, Mateusz Piotrowski, Richard Yao and Mariusz Zaborski.

This work was sponsored by Klara, Inc. and Wasabi Technology, Inc.

[1] Ahrens et al., (2005), *5096886 Write caching disks need mechanism to flush cache to physical media*, illumos/illumos-gate@fa9e4066 ↩