Wednesday, February 25. 2009

Insights into ZFS - today: The nature of writing things

So ... my last article was about the reading side of ZFS ... but whats about writing? As Ben correctly stated in his presentation about ZFS: Forget your good old UFS knowledge. ZFS is different. Especially writing stuff to disk is different to other filesystems. I want to concentrate on this part in article. I simplified this stuff a little bit for the ease of understanding. As usual the source code is the authorative source of informations about this stuff.
Sync writes vs. Async writesAt first: ZFS is a transactional file system. The way it writes data to the filesystem is a little bit more sophisticated. Okay, there are two different kinds of writes in an operating system syncronous and asynchronous writes. The difference: When a synchronous write comes back, the write is on disk. When a async write comes back, the write is in the cache of the system, waiting for it´s real write. The most famous generators of sync writes are databases. In order to confirm to the ACID paradigma, the database has to be sure, that data that was written to it, has to be written to a non-voilatile media. But there are others: A mail server has to use sync writes to be sure, no mail is lost in a power outage.

ZFS TransactionsThe ZFS Posix Layer (ZPL, the reason, why ZFS looks like a posix-compliant filesystem to you) encapsulate all write operations (either metadata or real data) into transactions. The ZPL opens a transaction, puts some write calls in it and commits it. But not every transaction is written to disk at once, they are collected into transaction groups.

ZFS Transaction GroupsAll writes in ZFS are handeled in transaction groups. Either syncronous writes or async writes are finally written this way to the hardisk. I will come back to the differences between both later in this blog articles. The concept of writing stuff to hard disk is largely based on the concept of the transaction groups. At any time there are at maximum three transaction groups in use per pool. The first one is the open group. This is the one, where the filesystem places new writes. The next one is the quiece group. You can add no new writes to it, it waits until all write operations to it have ceased. The last one is the sync group. This is the group where the data is finally written to disk. Working with a Transaction Group instead of single transactions gives you a nice advantage: You can bundle many transactions into a large write burst instead of many small ones thus saving IOPS from your IOPS budget.

The transaction group numberThroughout it´s lifecycle in the system every transaction group has an unique number. Together with the Uberblock this number plays an important role in recovering after a crash. There is an transaction number stored in the uberblock. You have to know, that the last operationg in every sync of the transaction group is the update of the Uberblock. It´s a single write block at the end. The uberblock is stored in a ring buffer of 128 blocks. The location of the uberblock is the transaction group number modulo 128. Finding the correct uberblock is easy. It´s the one with the highest transaction group number. Many of you will ask: And whats happening, when the power was interrupted while writing the ueberblock. Well, there is an additional safe guard. The uberblock is protected with a checksum. So the valid uberblock is the uberblock with the highest transaction block number with a correct checksum. Together with the copy-on-write (as you don´t overwrite data, you don´t harm existent data, and when the uberblock was written successfully to disk, the old state isn´t relevant anymore) and the ZIL this ensures an always consistent on-disk state.

Implementation in ZFSThere are actually three threads, that controls the writes in ZFS. There is the txg_timelimit_thread() thread. This fires up in an configurable interval, the default is 5 seconds. This threads tries to changes the open transaction group to the quiece state. Tries? Yes, there can be a most one TXG at a given time from every state in a single pool. Thus when the sync hasn´t completed, you can´t move a new txg from quiece to sync. When there is still a quieced TXG, you can´t move an open TXG to the quiece stage. In such a situation ZFS blocks new write calls to give the storage a change to catch up. Such an situation will happen, when you issue write operations to your file system at larger amount than your filesystem can handle.

Since last year the write throttling is implemented differently to slow down write intensive jobs a little bit (by a tick per write request) as soon the write rate surpasses a certain limit. Furthermore the system observes, if it can write outstanding data in the timlimit defined by txg_timelimit_thread(). When you really want to dig down into this issue and it´s solution i recomend Rochs blog article "The New ZFS Write Throttle".

Let´s assume the TXG has moved from open to quiece. This state is a very important one because of the transactional nature of ZFS. When an open TXG is moved to the quiece state, it´s possible that an opened transaction group contains transactions, that aren´t commited. The txg_quiesce_thread() waits until all transactions are commited and triggers the the txg_sync_thread(). This is finally the thread that writes the data on non-voilatile storage.

Where is the data stored from the Async writes until syncing?As long as the data isn´t synced to disk, all the data is kept in the Adaptive Replacement Cache. The ZFS dirties the pages in cache and writes them to disks. They are stored in memory. Thus when when you loose memory, all async changes are lost. But that´s why they are async and for writes that can´t accept such a loss, you should use sync writes.

Sync writesOkay, as you may have recoginized ZFS writes only in large bursts every few seconds. But how can we keep sync write semantics. We can´t keep them only in memory, as they would be only in volatile memory (DRAM) for a few seconds. To enable sync writes there is a special handling in the ZFS Intent Log, or short: ZIL.

The ZILThe ZIL keeps track of all write operations. Whenever you sync a filesystem or use the D_SYNC function call, the ZIL is written to a stable log device, for example on the disks itself. The ZIL contains enough informations to replay all the changes in case of a power failure. The ZIL list contains the modified data itself (in the case, it´s only a small amount of data) or contains a pointer to the data outside. The later one allows a neat trick: When you write a large heap of data, the data is written to your pool directly. The block pointer is delivered to the ZIL list and it´s part of the intent log. When the time comes, to put this to stable storage (thus integrating the stuff into the consistent on-disk-state of your filesystem, the data isn´t copied to from ZIL to the pool. ZFS just used the block pointer and simply uses the already written data of the blog.

Whenever the filesystem is unmounted, the ZIL on the stable media is fully processed to the filesystem and ZFS deletes it. When there is still a ZIL with active transactions at mount time, there must have been an unclean mount of the filesystem. So ZFS replays the transactions in the ZIL up to the state when the last sync write call came back or when the last successful sync of the pool took place .

Separated ZILThe separated ZIL in now an interesting twist to the story of handling the the ZIL. Instead of writing the ZIL blocks onto the data disks, you seperate them on a different media. This have two advantages: At first the sync writes doesn´t eat away IOPS from the IOPS budget of your storage (It can make even sense to use rotating rust disk for this tasks). Additionally you can use SSD for this task. The advantage: A sync write operation has to wait for the successful write to the disk. A magnetic rust disk has a much higher latency than a flash disk for example. Thus seperating away the ZIL to an SSD and keeping your normal storage on rotating rust, gives you both advantages: The incredible low write latency of SSD and the cheap and vast amounts of magnetic disk storage.

But one thing is important: The seperated ZIL just solves one problem: Latency of syncronous writes. It doesn´t help with insufficient bandwidth to your disk. It doesn´t help you with reads (well, that´s not entirely true, less IOPS for write leads to more IOPS for read, but that´s a corner case). But write latency is an issue everywhere: The writes of iSCSI targe to disk are syncronous per default. Most write operations of databases are synchronous. Metadata updates are synchronous. So in this cases an SSD can really help you. Or to be more exact: Whenever a seperated ZIL has a lower latency than the primary storage, a ZIL media is a good idea. So for local SAS,FC or SATA storage an SSD is a good idea, for remote storage (like iSCSI over WAN) one or two rotating rust disk can do similar things.

Conclusion ZFS goes a long way to make write operations as efficient as possible. The inside architecture of ZFS makes some technologies feasible, that are impossible or hard to implement with other filesystems. And most important: The always consistent state on disk is largely based on the transactions in conjunction with the copy-on-write (Yes, i know it´s more like redirect on write, Dirk)

```
Posted by Joerg Moellenkamp in English, Solaris at 14:37
```