ZPool CHECKPOINT

By Serapheim **Dimitropoulos**

In March of this year (2018), Alexander Motin (mav@freebsd.org) ported the Pool Checkpoint feature of OpenZFS from Illumos to FreeBSD. A pool checkpoint can be thought of as a "pool-wide snapshot" that captures the entire state of the pool at the time the checkpoint is created, allowing the user to revert the entire pool back to that state or discard it. The generic use cases are administrative tasks, like OS upgrades, that involve actions that change or destroy ZFS state and metadata. Examples of such actions are: enabling new pool features, changing properties of datasets, or destroying snapshots and filesystems. Before undertaking such actions, administrators can create a checkpoint of their pool and then apply their changes. If something goes wrong with the upgrade, the administrator can then rewind back to the checkpoint as if the actions had never been taken. In the same way a snapshot can help you return user data to a previous state, the checkpoint can help you return the ZFS pool to a previous state.

There are already two tutorials online demonstrating how to use this feature, and a block comment in the source code that gives a high-level overview of its implementation (see the References box). This article lies somewhere in the middle, giving a high-level description of what happens under the hood during each administrative operation.

Checkpointing a Pool

In ZFS we keep track of changes over time with Transaction Groups (aka TXGs). During each TXG, ZFS accumulates changes in memory and, when certain conditions are met, it syncs those changes to disk, then opens the next TXG. Each data block records the TXG during which it was created, called its birth TXG. Birth TXGs are important for the pool checkpoint feature because they help us differentiate between blocks created before and after a checkpoint. Finally, a TXG's last step when syncing to disk is writing the new uberblock to the beginning and end of each disk in an area called the ZFS label, which is different from the EFI label. Each uberblock is the root of the tree of the entire state of the pool for that TXG. Uberblocks are used during pool import as a starting point to find the latest version of all of the pool's data and metadata.

Whenever an administrator checkpoints a pool (with the "`zpool checkpoint`" command), ZFS copies the uberblock of the current TXG to an area within the pool's state called the Meta-Object Set (MOS). When this change is synced to disk, the uberblocks from subsequent TXGs reference the "checkpointed" uberblock through the MOS.

The Lifetime of a Block in a Checkpointed Pool

When a checkpoint exists, the process of allocating new blocks stays the same, but the process of freeing blocks is different. We can't free blocks that belong to the checkpoint, because we want to be able to rewind back to that point in time. At the same time, we can't stop freeing blocks entirely, because that would fill up the pool. Thus, every time we are about to free a block, we look at its birth TXG and compare it to the TXG of the "checkpointed" uberblock that we saved in the MOS. If the block was born at or before the TXG of the checkpointed uberblock, it means that the block is part of the checkpoint (i.e., it is referenced

by the checkpointed uberblock). These "checkpointed" blocks are never actually freed. Instead, we add their ranges to lists on-disk we call checkpoint spacemaps (there is one per `vdev`) and leave their segments marked as allocated so they are not reused by the pool. Blocks whose birth TXGs are after the checkpoint's TXG, are not part of the checkpointed state and can be freed normally.

Rewinding to the Checkpoint

If administrators want to rewind back to the checkpoint, all they need to do is to export and then reimport the pool with the rewind option ("`zpool import --rewind-to-checkpoint`"). In this case, the import process takes place as usual, but with one additional step. ZFS first looks at the current uberblock and from that it finds the checkpointed uberblock in the MOS. Then it uses the checkpointed uberblock instead of the current uberblock for the import process, effectively rewinding the pool back to the checkpointed state. Once the import process is done, the rewind is complete. The checkpoint spacemaps no longer exist as they were created after the checkpointed uberblock (which is now the current uberblock). For the same reason, there is no checkpoint uberblock in the MOS. This means that after rewinding there is no additional cleanup and the pool no longer has a checkpoint.

It is also possible to import a checkpoint read-only to access the pool in the state it existed at the time of the checkpoint without actually undoing all of the changes that have happened since the checkpoint was created. This can allow the administrator to recover specific data or a filesystem that was destroyed without rolling back the entire pool.

Discarding the Checkpoint

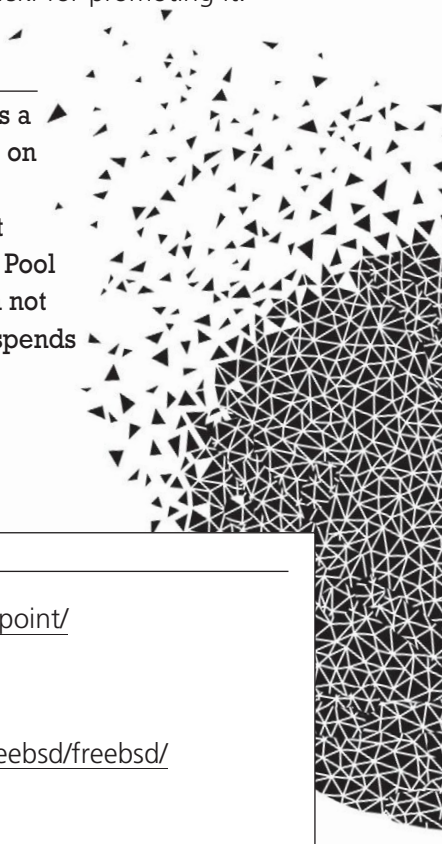
If administrators decide to get rid of the checkpoint, they run the discard command ("`zpool checkpoint --discard`"). The command instructs ZFS to get rid of the checkpointed uberblock from the MOS. At that point, the pool is considered to no longer have a checkpoint, which allows blocks to be freed normally regardless of their birth TXG. ZFS will also free all the previously recorded ranges from the checkpoint spacemaps—the blocks that we couldn't actually free because they were referenced by the checkpointed uberblock. The number of these blocks can be quite large depending on how long the checkpoint has existed and how many changes have been made to the pool. Freeing them all in a single TXG would be expensive. Instead, ZFS spawns a thread that frees them over the course of multiple TXGs by prefetching them into memory in chunks and freeing one chunk per TXG.

Acknowledgments

The development of this feature would not have been possible without the help of my colleagues from Delphix, especially Dan Kimmel, who started the initial prototype with me, and Matt Ahrens for guiding me every step of the way. I'd also like to thank Alexander Motin for porting the feature to FreeBSD, and Marius Zaborski for promoting it. •

.

Serapheim Dimitropoulos is a software engineer working on ZFS at Delphix. His main contributions to the project are the Log Spacemap and Pool Checkpoint features. When not programming, Serapheim spends his time playing soccer, salsa dancing, and playing classical guitar.



REFERENCES

Serapheim's ***ZPool Checkpoint*** tutorial. http://sdimitro.github.io/post/zpool-checkpoint/

Marius's ***ZPool Checkpoint*** tutorial. http://oshogbo.vexillium.org/blog/46/

Implementation Overview Block Comment in the source. https://github.com/freebsd/freebsd/blob/master/sys/cddl/contrib/opensolaris/uts/common/fs/zfs/spa_checkpoint.c