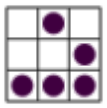


Aaron Toponce

{ 2012.04.17 }

Install ZFS on Debian GNU/Linux

Table of Contents

UPDATE (May 06, 2012): I apologize for mentioning it supports encryption. Pool version 28 is the latest source that the Free Software community has. Encryption was not added until pool version 30. So, encryption is not supported natively with the ZFS on Linux project. However, you can use LUKS containers underneath, or you can use Ecryptfs for the entire filesystem, which would still give you all the checksum, scrubbing and data integrity benefits of ZFS. Until Oracle gets their act together, and releases the current sources of ZFS, crypto is not implemented.

Quick post on installing ZFS as a kernel module, not FUSE, on Debian GNU/Linux. The documents already exist for getting this going, I'm just hoping to spread this to a larger audience, in case you are unaware that it exists.

First, the Lawrence Livermore National Laboratory has been working on porting the native Solaris ZFS source to the Linux kernel as a kernel module. So long as the project remains under contract by the Department of Defense in the United States, I'm confident there will be continuous updates. You can track the progress of that porting at http://zfsonlinux.org.

UPDATE (May 05, 2013): I've updated the installation instructions. The old instructions included downloading the source and installing from there. At the time, that was all that was available. Since then, the ZFS on Linux project has created a proper Debian repository that you can use to install ZFS. Here is how you would do that:

```
$ su -
# wget http://archive.zfsonlinux.org/debian/pool/main/z/zfsonlinux/zfsonlinux_2%7Ewheezy_all.deb
# dpkg -i zfsonlinux_2~wheezy_all.deb
# apt-get update
# apt-get install debian-zfs
```

And that's it!

If you're running Ubuntu, which I know most of you are, you can install the packages from the Launchpad PPA https://launchpad.net/~zfs-native.

UPDATE (May 05, 2013): The following instructions may not be relevant for fixing the manpages. If they are, I've left them in this post, just struck out.

~~**A word of note:** the manpages get installed to /share/man/. I found this troubling. You can modify your $MANPATH variable to include /share/man/man8/, or by creating symlinks, which is the approach I took:~~

```
~~# cd /usr/share/man/man8/~~
~~# ln -s /share/man/man8/zdb.8 zdb.8~~
~~# ln -s /share/man/man8/zfs.8 zfs.8~~
~~# ln -s /share/man/man8/zpool.8 zpool.8~~
```

Now, make your zpool, and start playing:

```
$ sudo zpool create test raidz sdd sde sdf sdg sdh sdi
```

It is stable enough to run a ZFS root filesystem on a GNU/Linux installation for your workstation as something to play around with. It is copy-on-write, supports compression, deduplication, file atomicity, off-disk caching, ~~encryption,~~ and much more. At this point, unfortunately, I'm convinced that ZFS as a Linux kernel module will become "stable" long before Btrfs will be stable in the mainline kernel. Either way, it doesn't matter to me. Both are Free Software, and both provide the long needed features we've needed with today's storage needs. Competition is healthy, and I love having choice. Right now, that choice might just be ZFS.

Posted by Aaron Toponce on Tuesday, April 17, 2012, at 2:10 pm.

Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 22 } Comments

1. OdyX | April 18, 2012 at 1:44 am | Permalink

   How happens that zfsonlinux is not yet packaged in Debian ?

   As far as I read http://zfsonlinux.org/faq.html#WhatAboutTheLicensingIssue , Debian could distribute the zfsonlinux source and build it on user machines using dkms for example, right?

2. cristalinox | May 1, 2012 at 1:28 am | Permalink

   sudo dpkg -i *_amd64.deb
   before
   Now do the same for ZFS ?????

3. Aaron Toponce | May 1, 2012 at 2:36 pm | Permalink

   According to the post:

   $ sudo dpkg -i ~/src/{spl,zfs}/*.deb

   That should take care of both the SPL and ZFS packages.

4. Gary | May 4, 2012 at 3:48 pm | Permalink

   You can't build zfs until spl devel is installed.... at least I couldn't 🙂

5. Rudd-O | May 6, 2012 at 2:55 am | Permalink

   I wrote a guide to get ZFS (and a root file system backed by it) on Fedora.

   http://rudd-o.com/linux-and-free-software/installing-fedora-on-top-of-zfs/index

6. DaFresh | May 6, 2012 at 10:06 am | Permalink

   Odyx, there is a licencing problem, CDDL for ZFS, GPL for Debian/Linux, which are not compatible.

7. Daniel Mccumllam | May 16, 2012 at 1:36 am | Permalink

   Brilliant update information pal. Sounds like you install ZFS on Debian GNU/Linux. I hope I can install soon for mentioning it supports encryption. Thanks!

8. Paul Nienaber | June 25, 2012 at 11:18 am | Permalink

   Dude, './configure --mandir=/usr/local/share/man'. Problem solved. Thanks for the heads-up that it's going to do that 🙂

9. Jonas | July 8, 2012 at 8:39 am | Permalink

   Very useful guide! I can now import my old FreeNAS ZFS disks on my new Debian server. Awesome.

10. Martin | December 8, 2012 at 3:22 pm | Permalink

    Thank You very much. Short and sweet. I had zfs-fuse running and was quite happy but always wanted to change to the kernel module. With Your post it worked like a charm. Make sure zfs-fuse is deinstalled, because You get a conflict on installing the zfs-Packages.

    Regards,

    Martin

    Debian Sid Distro Aptosid w. 3.6.9

11. bodhi | December 17, 2012 at 1:04 am | Permalink

    Thanks for the instruction. I've downloaded the source and compiled spl and zfs for armv5. Compilation went OK, no problem. But I got this error in building the deb (make deb):

    spl-0.6.0-rc12.armv5tel.rpm is for architecture armv5tel ; the package cannot be built on this system.

    Any idea why?

12. Aaron Toponce | December 17, 2012 at 7:13 am | Permalink

    Probably because it's using "alien" to covert the RPM to a DEB. I would file a bug report.

13. bodhi | December 18, 2012 at 1:11 am | Permalink

    Thanks for the response Aaron. Glad to know you think it's related to "alien" (not armv5 arch).

14. David Anderson | December 18, 2012 at 8:37 am | Permalink

Can I install a non-root system on 12.10? (because the grub issue is not yet fixed for 12.10)

15. Aaron Toponce | December 18, 2012 at 12:28 pm | Permalink

    I'm not sure what you mean. Do you mean that you want to install ext4 (or something else) on / and ZFS elsewhere? In my case, I use ext4 for /, due to the GRUB problems you mention, and use ZFS for /home/, /var/cache/ and /var/log/ on my workstation. Of course, on our storage servers at work, ZFS is usually mounted to /backup/ or something else based on its function.

16. David Anderson | December 18, 2012 at 2:19 pm | Permalink

    Aaron, that IS what I meant... The repo is for 12.04 only at this time.

17. David E. Anderson | January 2, 2013 at 9:48 pm | Permalink

    on 12.10 I had to add zlib1g-dev and uuid-dev to the pre-requisite installed software

18. Dju | June 9, 2013 at 9:55 am | Permalink

    hi

    thanks for all the explainations !
    i was planning on setting up my filer @home with 3/4 disks (proliant microserver, with Debian wheezy)
    Now i've spent a few hours reading this article, i will definitely use ZFS 🙂

    By the way, about the bug you're talking about when sharing a dataset with samba, the issue is not in ZFS but samba, i've known it for a long time...
    Basically, when accessing the share from a remote computer with a user/password, samba will refuse the access, because the local user on your server is NOT included in samba users list....
    To do that, yout need to add it by typing
    smbpassword -a your_user
    then enter the password twice. then it will work 😉

19. Ante | September 1, 2013 at 2:40 am | Permalink

    Hi,

    I have been following this instructions but at the last step I get following:
    Package debian-zfs is not available, but is referred by another package.
    This may mean that the package is missing, has been obsoleted, or is only available from another source

    E: Package 'debian-zfs' has no installation candidate

20. Eric | June 13, 2014 at 2:24 pm | Permalink

    0.6.2 to 0.6.3 in Debian Wheezy:
    in /etc/default/zfs
    I had to change this setting from Yes to No in order for the automatic mount to work:
    USE_DISK_BY_ID='no'

21. Chin Huat Tan | June 20, 2015 at 11:32 pm | Permalink

    Hi Aaron, will you be updating these documentations as new versions are introduced? Will current documentation still be valid for version 6.4?

22. CROW KNOWS | July 17, 2017 at 12:47 pm | Permalink

Many thanks for the outstanding ZFS write-up (it's still relevant in 2017)!

{ 27 } Trackbacks

1. Установка ZFS на Debian GNU/Linux « Иркутский LUG | April 18, 2012 at 7:40 pm | Permalink

   [...] Оригинал статьи: тут Поделиться ссылкой:Facebookпо электронной [...]

2. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 4, 2012 at 6:00 am | Permalink

   [...] about how you can administer your ZFS filesystems and pools. You should start first by reading how to get ZFS installed into your GNU/Linux system here on this blog, then continue with this [...]

3. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 13, 2012 at 5:54 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

4. Aaron Toponce : ZFS Administration, Part VI- Scrub and Resilver | December 13, 2012 at 5:56 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

5. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | December 13, 2012 at 5:57 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

6. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | December 13, 2012 at 5:57 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

7. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | December 13, 2012 at 5:58 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

8. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 13, 2012 at 5:58 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

9. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | December 13, 2012 at 6:01 am | Permalink

   [...] Install ZFS on Debian GNU/Linux [...]

10. Aaron Toponce : ZFS Administration, Part IX- Copy-on-write | December 14, 2012 at 6:01 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

11. Aaron Toponce: ZFS Administration, Part IX- Copy-on-write - Bartle Doo | December 14, 2012 at 11:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux VDEVs RAIDZ The ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and Caveats [...]

12. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | December 17, 2012 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

13. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | December 18, 2012 at 6:01 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

14. Aaron Toponce: ZFS Administration, Part XI- Compression and Deduplication - Bartle Doo | December 18, 2012 at 4:40 pm | Permalink

    [...] Install ZFS on Debian GNU/Linux VDEVs RAIDZ The ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and Caveats [...]

15. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | December 19, 2012 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

16. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | December 20, 2012 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

17. Aaron Toponce: ZFS Administration, Part XIII- Sending and Receiving Filesystems - Bartle Doo | December 20, 2012 at 2:32 pm | Permalink

    [...] Install ZFS on Debian GNU/Linux VDEVs RAIDZ The ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and Caveats [...]

18. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | December 21, 2012 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

19. Aaron Toponce: ZFS Administration, Part XIV- ZVOLS - Bartle Doo | December 21, 2012 at 11:02 am | Permalink

    [...] Install ZFS on Debian GNU/Linux VDEVs RAIDZ The ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and Caveats [...]

20. Aaron Toponce : ZFS Administration, Part XV- iSCSI, NFS and Samba | December 31, 2012 at 6:01 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

21. Aaron Toponce : ZFS Administration, Part XVI- Getting and Setting Properties | January 2, 2013 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

22. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | January 3, 2013 at 6:02 am | Permalink

[...] Install ZFS on Debian GNU/Linux [...]

23. artodeto's blog about coding, politics and the world | February 12, 2013 at 3:05 pm | Permalink

    zfsonlinux and kernel 3.7.7-1-ARCH - not right now...

    A new kernel (3.7.7) was released today for the arch linux. Currently, the zfsonlinux in the aur was not adapted on it. I have updated my zfsonlinux automake tool with a more secure option. Before you are going to rebuild the zfs modules, the tool now...

24. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:02 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

25. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | May 9, 2013 at 6:00 am | Permalink

    [...] Install ZFS on Debian GNU/Linux [...]

26. zfs on linux 0.6.2 released | 阿木图技术博客 | August 25, 2013 at 5:47 am | Permalink

    […] 另外，发现ZoL上面的一个文档不错，包含了ZFS一些内部原理，比如ARC、COW等，值得推荐。 此条目发表在 […]

27. ZFS Stammtisch - Seite 163 | October 7, 2013 at 1:39 am | Permalink

    […] von ZFS verdeutlicht. Mir hilft das gerade sehr, mich in die Materie einzuarbeiten. Hier der Link: Aaron Toponce : Install ZFS on Debian GNU/Linux […]

Aaron Toponce

{ 2012.12.04 }

ZFS Administration, Part I- VDEVs

Table of Contents

So, I've blogged a few times randomly about getting ZFS on GNU/Linux, and it's been a hit. I've had plenty of requests for blogging more. So, this will be the first in a long series of posts about how you can administer your ZFS filesystems and pools. You should start first by reading how to get ZFS installed into your GNU/Linux system here on this blog, then continue with this post.

Virtual Device Introduction

To start, we need to understand the concept of virtual devices, or VDEVs, as ZFS uses them internally extensively. If you are already familiar with RAID, then this concept is not new to you, although you may not have referred to it as "VDEVs". Basically, we have a meta-device that represents one or more physical devices. In Linux software RAID, you might have a "/dev/md0" device that represents a RAID-5 array of 4 disks. In this case, "/dev/md0" would be your "VDEV".

There are seven types of VDEVs in ZFS:

1. disk (default)- The physical hard drives in your system.
2. file- The absolute path of pre-allocated files/images.
3. mirror- Standard software RAID-1 mirror.
4. raidz1/2/3- Non-standard distributed parity-based software RAID levels.
5. spare- Hard drives marked as a "hot spare" for ZFS software RAID.
6. cache- Device used for a level 2 adaptive read cache (L2ARC).
7. log- A separate log (SLOG) called the "ZFS Intent Log" or ZIL.

It's important to note that VDEVs are always dynamically striped. This will make more sense as we cover the commands below. However, suppose there are 4 disks in a ZFS stripe. The stripe size is calculated by the number of disks and the size of the disks in the array. If more disks are added, the stripe size can be adjusted as needed for the additional disk. Thus, the dynamic nature of the stripe.

Some zpool caveats

I would be amiss if I didn't meantion some of the caveats that come with ZFS:

- Once a device is added to a VDEV, it cannot be removed.
- You cannot shrink a zpool, only grow it.
- RAID-0 is faster than RAID-1, which is faster than RAIDZ-1, which is faster than RAIDZ-2, which is faster than RAIDZ-3.
- Hot spares are not dynamically added unless you enable the setting, which is off by default.
- A zpool will not dynamically rezise when larger disks fill the pool unless you enable the setting BEFORE your first disk replacement, which is off by default.
- A zpool will know about "advanced format" 4K sector drives IF AND ONLY IF the drive reports such.

- Deduplication is EXTREMELY EXPENSIVE, will cause performance degredation if not enough RAM is installed, and is pool-wide, not local to filesystems.
- On the other hand, compression is EXTREMELY CHEAP on the CPU, yet it is disabled by default.
- ZFS suffers a great deal from fragmentation, and full zpools will "feel" the performance degredation.
- ZFS suports encryption natively, but it is NOT Free Software. It is proprietary copyrighted by Oracle.

For the next examples, we will assume 4 drives: /dev/sde, /dev/sdf, /dev/sdg and /dev/sdh, all 8 GB USB thumb drives. Between each of the commands, if you are following along, then make sure you follow the cleanup step at the end of each section.

A simple pool

Let's start by creating a simple zpool wyth my 4 drives. I could create a zpool named "tank" with the following command:

```
# zpool create tank sde sdf sdg sdh
```

In this case, I'm using four disk VDEVs. Notice that I'm not using full device paths, although I could. Because VDEVs are always dynamically striped, this is effectively a RAID-0 between four drives (no redundancy). We should also check the status of the zpool:

```
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          sde       ONLINE       0     0     0
          sdf       ONLINE       0     0     0
          sdg       ONLINE       0     0     0
          sdh       ONLINE       0     0     0

errors: No known data errors
```

Let's tear down the zpool, and create a new one. Run the following before continuing, if you're following along in your own terminal:

```
# zpool destroy tank
```

A simple mirrored zpool

In this next example, I wish to mirror all four drives (/dev/sde, /dev/sdf, /dev/sdg and /dev/sdh). So, rather than using the disk VDEV, I'll be using "mirror". The command is as follows:

```
# zpool create tank mirror sde sdf sdg sdh
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sde     ONLINE       0     0     0
            sdf     ONLINE       0     0     0
            sdg     ONLINE       0     0     0
            sdh     ONLINE       0     0     0

errors: No known data errors
```

Notice that "mirror-0" is now the VDEV, with each physical device managed by it. As mentioned earlier, this would be analogous to a Linux software RAID "/dev/md0" device representing the four physical devices. Let's now clean up our pool, and create another.

```
# zpool destroy tank
```

Nested VDEVs

VDEVs can be nested. A perfect example is a standard RAID-1+0 (commonly referred to as "RAID-10"). This is a stripe of mirrors. In order to specify the nested VDEVs, I just put them on the command line in order (emphasis mine):

```
# zpool create tank mirror sde sdf mirror sdg sdh
# zpool status
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sde     ONLINE       0     0     0
            sdf     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            sdg     ONLINE       0     0     0
            sdh     ONLINE       0     0     0

errors: No known data errors
```

The first VDEV is "mirror-0" which is managing /dev/sde and /dev/sdf. This was done by calling "mirror sde sdf". The second VDEV is "mirror-1" which is managing /dev/sdg and /dev/sdh. This was done by calling "mirror sdg sdh". Because VDEVs are always dynamically striped, "mirror-0" and "mirror-1" are striped, thus creating the RAID-1+0 setup. Don't forget to cleanup before continuing:

```
# zpool destroy tank
```

File VDEVs

As mentioned, pre-allocated files can be used fer setting up zpools on your existing ext4 filesystem (or whatever). It should be noted that this is meant entirely for testing purposes, and not for storing production data. Using files is a great way to have a sandbox, where you can test compression ratio, the size of the deduplication table, or other things without actually committing production data to it. When creating file VDEVs, you cannot use relative paths, but must use absolute paths. Further, the image files must be preallocated, and not sparse files or thin provisioned. Let's see how this works:

```
# for i in {1..4}; do dd if=/dev/zero of=/tmp/file$i bs=1G count=4 &> /dev/null; done
# zpool create tank /tmp/file1 /tmp/file2 /tmp/file3 /tmp/file4
# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:

        NAME           STATE     READ WRITE CKSUM
        tank           ONLINE       0     0     0
          /tmp/file1   ONLINE       0     0     0
          /tmp/file2   ONLINE       0     0     0
          /tmp/file3   ONLINE       0     0     0
          /tmp/file4   ONLINE       0     0     0

errors: No known data errors
```

In this case, we created a RAID-0. We used preallocated files using /dev/zero that are each 4GB in size. Thus, the size of our zpool is 16 GB in usable space. Each file, as with our first example using disks, is a VDEV. Of course, you can treat the files as disks, and put them into a mirror configuration, RAID-1+0, RAIDZ-1 (coming in the next post), etc.

```
# zpool destroy tank
```

Hybrid pools

This last example should show you the complex pools you can setup by using different VDEVs. Using our four file VDEVs from the previous example, and our four disk VDEVs /dev/sde through /dev/sdh, let's create a hybrid pool with cache and log drives. Again, I emphasized the nested VDEVs for clarity:

```
# zpool create tank mirror /tmp/file1 /tmp/file2 mirror /tmp/file3 /tmp/file4 log mirror sde sdf cache sdg sdh
# zpool status tank
  pool: tank
 state: ONLINE
```

```
 scan: none requested
config:

        NAME            STATE     READ WRITE CKSUM
        tank            ONLINE       0     0     0
          mirror-0      ONLINE       0     0     0
            /tmp/file1  ONLINE       0     0     0
            /tmp/file2  ONLINE       0     0     0
          mirror-1      ONLINE       0     0     0
            /tmp/file3  ONLINE       0     0     0
            /tmp/file4  ONLINE       0     0     0
        logs
          mirror-2      ONLINE       0     0     0
            sde         ONLINE       0     0     0
            sdf         ONLINE       0     0     0
        cache
          sdg           ONLINE       0     0     0
          sdh           ONLINE       0     0     0

errors: No known data errors
```

There's a lot going on here, so let's disect it. First, we created a RAID-1+0 using our four preallocated image files. Notice the VDEVs "mirror-0" and "mirror-1", and what they are managing. Second, we created a third VDEV called "mirror-2" that actually is not used for storing data in the pool, but is used as a ZFS intent log, or ZIL. We'll cover the ZIL in more detail in another post. Then we created two VDEVs for caching data called "sdg" and "sdh". The are standard disk VDEVs that we've already learned about. However, they are also managed by the "cache" VDEV. So, in this case, we've used 6 of the 7 VDEVs listed above, the only one missing is "spare".

Noticing the indentation will help you see what VDEV is managing what. The "tank" pool is comprised of the "mirror-0" and "mirror-1" VDEVs for long-term persistent storage. The ZIL is magaged by "mirror-2", which is comprised of /dev/sde and /dev/sdf. The read-only cache VDEV is managed by two disks, /dev/sdg and /dev/sdh. Neither the "logs" nor the "cache" are long-term storage for the pool, thus creating a "hybrid pool" setup.

```
# zpool destroy tank
```

Real life example

In production, the files would be physical disk, and the ZIL and cache would be fast SSDs. Here is my current zpool setup which is storing this blog, among other things:

```
# zpool status pool
  pool: pool
 state: ONLINE
 scan: scrub repaired 0 in 2h23m with 0 errors on Sun Dec  2 02:23:44 2012
config:

        NAME                                            STATE     READ WRITE CKSUM
        pool                                            ONLINE       0     0     0
          raidz1-0                                      ONLINE       0     0     0
            sdd                                         ONLINE       0     0     0
            sde                                         ONLINE       0     0     0
            sdf                                         ONLINE       0     0     0
            sdg                                         ONLINE       0     0     0
        logs
          mirror-1                                      ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part1  ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-X5RG0EIY7MN7676K-part1  ONLINE       0     0     0
        cache
          ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part2    ONLINE       0     0     0
          ata-OCZ-REVODRIVE_OCZ-X5RG0EIY7MN7676K-part2    ONLINE       0     0     0

errors: No known data errors
```

Notice that my "logs" and "cache" VDEVs are OCZ Revodrive SSDs, while the four platter disks are in a RAIDZ-1 VDEV (RAIDZ will be discussed in the next post). However, notice that the name of the SSDs is "ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part1", etc. These are found in /dev/disk/by-id/. The reason I chose these instead of "sdb" and "sdc" is because the cache and log devices don't necessarily store the same ZFS metadata. Thus, when the pool is being created on boot, they may not come into the pool, and could be missing. Or, the motherboard may assign the drive letters in a different order. This isn't a problem with the main pool, but is a big problem on GNU/Linux with logs and cached devices. Using the device name under /dev/disk/by-id/ ensures greater persistence and uniqueness.

Also do notice the simplicity in the implementation. Consider doing something similar with LVM, RAID and ext4. You would need to do the following:

```
# mdadm -C /dev/md0 -l 0 -n 4 /dev/sde /dev/sdf /dev/sdg /dev/sdh
# pvcreate /dev/md0
# vgcreate /dev/md0 tank
# lvcreate -l 100%FREE -n videos tank
# mkfs.ext4 /dev/tank/videos
# mkdir -p /tank/videos
# mount -t ext4 /dev/tank/videos /tank/videos
```

The above was done in ZFS (minus creating the logical volume, which will get to later) with one command, rather than seven.

Conclusion

This should act as a good starting point for getting the basic understanding of zpools and VDEVs. The rest of it is all downhill from here. You've made it over the "big hurdle" of understanding how ZFS handles pooled storage. We still need to cover RAIDZ levels, and we still need to go into more depth about log and cache devices, as well as pool settings, such as deduplication and compression, but all of these will be handled in separate posts. Then we can get into ZFS filesystem datasets, their settings, and advantages and disagvantages. But, you now have a head start on the core part of ZFS pools.

Posted by Aaron Toponce on Tuesday, December 4, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 16 } Comments

1. Wayne | December 4, 2012 at 7:21 am | Permalink

   Thanks

2. Adam Stovicek | December 4, 2012 at 2:33 pm | Permalink

   Thank-you for getting into the meat and potatoes of managing ZFS pools. I've been tinkering with the idea of playing around with a FreeBSD/FreeNAS installation for file server duties but haven't found the free time. One hurdle has been understanding ZFS, which doesn't seem all to difficult, but having it explained in your usual style of straightforward simplicity helps out a lot.

3. Aaron Toponce | December 4, 2012 at 4:21 pm | Permalink

   No problem Adam. I have about 20 more posts scheduled, so hopefully this helps. Of course, if anything doesn't make sense, let me know.

4. boneidol | December 28, 2012 at 7:20 pm | Permalink

   "Because VDEVs are always dynimacally striped, " <- trivial typo

5. Aaron Toponce | December 29, 2012 at 6:24 am | Permalink

   Fixed. Thanks!

6. Ruurd | April 16, 2013 at 3:36 am | Permalink

   Good article. Thanks for the clear explanation.

7. Dan | May 2, 2013 at 6:54 am | Permalink

   Thanks for writing this up, definitely helping me get my feet wet with ZFS 🙂 Slight nitpicking - your comparison with the Linux software RAID includes the creation of the mount point and actually mounting it, which the ZFS pool creation command does not.

8. Ed Cutler | June 1, 2013 at 8:29 pm | Permalink

   Thanks for the write-up Aaron! I'm pretty new to zfs, and was wondering. At the beginning of your write-up, you mention VDEVs as being like a RAID array, and then you go into caveats for zpools. Did I miss something? I come from a NetApp background so I was trying to relate this to their terminology. Is a zpool like a logical volume made up of VDEVs? In your examples, each VDEV is a single disk, but can you setup each VDEV as a RAID array and then add the VDEVs to a zpool? Thanks ahead for the clarification.

9. Aaron Toponce | June 2, 2013 at 8:19 am | Permalink

   A VDEV is one of 7 things: disk, file, mirror, raidz1,2,3, spare, cache, & log. If you use 'mirror' for 2 disks, as an example, then you have built a pool with one VDEV. If you wish to add two more disks as another mirror, then you have added an additional VDEV to the pool. So to answer your question, yes, a zpool is a logical volume pool made up of one or more VDEVs.

10. Ryan | September 15, 2013 at 4:16 pm | Permalink

    I ran into a couple problems in creating my zpool. First, I used the command:

    zpool create tank raidz2 -o ashift=12 /dev/sda /dev/sdb /dev/sdc /dev/sdd /dev/sde /dev/sdf

    Which returned the following message:

    invalid vdev specification
    use '-f' to override the following errors:
    /dev/sdc does not contain an EFI label but it may contain partition
    information in the MBR.

    At first, I wasn't sure what this meant, but I noticed that it complained about /dev/sdc, but it seemed to be ok with /dev/sda and /dev/sdb, both of which were formatted using gdisk. So I assumed this message was a warning that the disks had other (non gpt) partition information. Is that correct? I used the -f option and it created the zpool just fine.

    Although it isn't necessary to do so, I decided to use the device name under /dev/disk/by-id/ for each. These names weren't available prior to creating the zpool, but they are now, so I destroyed the zpool and created a new one using the new names. I chose to do this because I've also run into problems with the devices getting renamed, which makes it tricky when it comes time to replace a failed drive. I'm labeling the disks with their /dev/disk/by-id names, so that problem won't come up.

11. Brian Hanna | December 20, 2013 at 2:31 pm | Permalink

    Great writeup, thanks much. Under "Real Life Example", I notice you don't have the zpool create command, you go right to zpool status. Looking forward to more.

12. Aaron Toponce | December 21, 2013 at 7:57 am | Permalink

    Yes. For that, the pool already exists, and is "in production". So I'm just showing you the status of the pool, rather than how to create it. At this point in the post, you should be familiar enough with how to create ZFS pools.

13. Andreas | February 27, 2014 at 4:50 pm | Permalink

    "Once a device is added to a VDEV, it cannot be removed."

    This is not true, it's possible to change mirror devices: 'zpool attach' can be used to add drives to a mirror, or to create a mirror from two single drives, and 'zpool split' can be used to split a disk off a mirror.

    You should really change this, since zfsonlinux.org links to you as documentation.

14. Renjith | February 21, 2017 at 11:17 am | Permalink

    What makes more sense for a protected & performing 30TB POOL with reasonable expansion capability.

    3x3TB ZFS1 vdevs X 6
    OR
    6x3TB ZFS2 vdevs X 3

15. asmo | June 26, 2017 at 5:27 pm | Permalink

Will a hot spare replace a SSD in a cache-VDEV as well? If so, how do I tell ZFS to use the hot spare only for disks in the "payload-VDEV"? These are named "tank" and "pool" above. Thanks in advance!

16. Abdollah | January 11, 2018 at 5:06 am | Permalink

    Is possible to mirror two raidz(1-3) together?
    for example 4 drive raidz1 with name of pool1 mirrored with another 4 drive raidz1 with name of pool2?

{ 19 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 5, 2012 at 6:00 am | Permalink

   [...] The previous post introduced readers to the concept of VDEVs with ZFS. This post continues the topic discusing the RAIDZ VDEVs in great detail. [...]

2. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 13, 2012 at 5:55 am | Permalink

   [...] VDEVs [...]

3. Aaron Toponce : Install ZFS on Debian GNU/Linux | December 13, 2012 at 6:00 am | Permalink

   [...] VDEVs [...]

4. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | December 13, 2012 at 6:04 am | Permalink

   [...] VDEVs [...]

5. ZFS administration (1) « 0ddn1x: tricks with *nix | December 18, 2012 at 11:16 am | Permalink

   [...] http://pthree.org/2012/12/04/zfs-administration-part-i-vdevs/ [...]

6. Aaron Toponce : ZFS Administration, Part VI- Scrub and Resilver | December 18, 2012 at 12:35 pm | Permalink

   [...] VDEVs [...]

7. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | December 20, 2012 at 8:05 am | Permalink

   [...] VDEVs [...]

8. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | December 20, 2012 at 8:06 am | Permalink

   [...] VDEVs [...]

9. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | December 20, 2012 at 8:08 am | Permalink

   [...] VDEVs [...]

10. Aaron Toponce : ZFS Administration, Part XV- iSCSI, NFS and Samba | January 1, 2013 at 11:04 am | Permalink

    [...] VDEVs [...]

11. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | January 7, 2013 at 9:17 pm | Permalink

    [...] VDEVs [...]

12. Aaron Toponce : ZFS Administration, Part XVI- Getting and Setting Properties | January 7, 2013 at 9:18 pm | Permalink

    [...] VDEVs [...]

13. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | January 7, 2013 at 9:19 pm | Permalink

    [...] VDEVs [...]

14. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | January 7, 2013 at 9:19 pm | Permalink

[...] VDEVs [...]

15. Aaron Toponce : ZFS Administration, Part IX- Copy-on-write | January 7, 2013 at 9:24 pm | Permalink

    [...] VDEVs [...]

16. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | March 20, 2013 at 12:37 pm | Permalink

    [...] VDEVs [...]

17. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:09 am | Permalink

    [...] VDEVs [...]

18. PC-BSD: souborový systém ZFSÚvěr | Úvěr | May 26, 2013 at 4:19 pm | Permalink

    [...] Mnoho informací o ZFS je samozřejmě možné najít na webu, například u Oracle nebo Aarona Toponce. [...]

19. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | July 8, 2013 at 10:07 pm | Permalink

    […] VDEVs […]

Aaron Toponce

{ 2012.12.05 }

ZFS Administration, Part II- RAIDZ

Table of Contents

The previous post introduced readers to the concept of VDEVs with ZFS. This post continues the topic discusing the RAIDZ VDEVs in great detail.

Standards Parity RAID

To understand RAIDZ, you first need to understand parity-based RAID levels, such as RAID-5 and RAID-6. Let's discuss the standard RAID-5 layout. You need a minimum of 3 disks for a proper RAID-5 array. On two disks, the data is striped. A parity bit is then calculated such than the XOR of all three stripes in the set is calculated to zero. The parity is then written to disk. This allows you to suffer one disk failure, and recalculate the data. Further, in RAID-5, no single disk in the array is dedicated for the parity data. Instead, the parity is distributed throughout all of the disks. Thus, any disk can fail, and the data can still be restored.

However, we have a problem. Suppose that you write the data out in the RAID-5 stripe, but a power outtage occurs before you can write the parity. You now have inconsistent data. Jeff Bonwick, the creator of ZFS, refers to this as a "RAID-5 write hole". In reality, it's a problem, no matter how small, for all parity-based RAID arrays. If there exists any possibility that you can write the data blocks without writing the parity bit, then we have the "write hole". What sucks, is the software-based RAID is not aware that a problem exists. Now, there are software work-arounds to identify that the parity is inconsistent with the data, but they're slow, and not reliable. As a result, software-based RAID has fallen out of favor with storage administrators. Rather, expensive (and failure-prone) hardware cards, with battery backups on the card, have become commonplace.

There is also a big performance problem to deal with. If the data being written to the stripe is smaller than the stripe size, then the data must be read on the rest of the stripe, and the parity recalculated. This causes you to read and write data that is not pertinent to the application. Rather than reading only live, running data, you spend a great deal of time reading "dead" or old data. So, as a result, expensive battery-backed NVRAM hardware RAID cards can hide this latency from the user, while the NVRAM buffer fills working on this stripe, until it's been flushed to disk.

In both cases, the RAID-5 write hole, and writing data to disk that is smaller than the stripe size, the atomic transactional nature of ZFS does not like the hardware solutions, as it's impossible, and does not like existing software solutions as it opens up the possibility of corrupted data. So, we need to rethink parity-based RAID.

ZFS RAIDZ

Enter RAIDZ. Rather than the stripe width be statically set at creation, the stripe width is dynamic. Every block transactionally flushed to disk is its own stripe width. Every RAIDZ write is a full stripe write. Further, the parity bit is flushed with the stripe simultaneously, completely eliminating the RAID-5 write hole. So, in the event of a power failure, you either have the latest flush of data, or you don't. But, your disks will not be inconsistent.

RAID-5			
A1	A2	A3	Ap
B1	B2	Bp	B3
C1	Cp	C2	C3
Dp	D1	D2	D3
E1	E2	E3	Ep
F1	F2	Fp	F3
G1	Gp	G2	G3
HP	H1	H2	H3

RAID-Z1			
A1	A2	A3	Ap
A4	A5	A6	Ap'
B1	B2	Bp	C1
C2	C3	Cp	D1
Dp	E1	E2	E3
Ep	E4	Ep'	F1
Fp	G1	G2	G3
Gp	H1	H2	Hp

Demonstrating the dynamic stripe size of RAIDZ

There's a catch however. With standardized parity-based RAID, the logic is as simple as "every disk XORs to zero". With dynamic variable stripe width, such as RAIDZ, this doesn't work. Instead, we must pull up the ZFS metadata to determine RAIDZ geometry on every read. If you're paying attention, you'll notice the impossibility of such if the filesystem and the RAID are separate products; your RAID card knows nothing of your filesystem, and vice-versa. This is what makes ZFS win.

Further, because ZFS knows about the underlying RAID, performance isn't an issue unless the disks are full. Reading filesystem metadata to construct the RAID stripe means only reading live, running data. There is no worry about reading "dead" data, or unallocated space. So, metadata traversal of the filesystem can actually be faster in many respects. You don't need expensive NVRAM to buffer your write, nor do you need it for battery backup in the event of RAID write hole. So, ZFS comes back to the old promise of a "Redundant Array of Inexpensive Disks". In fact, it's highly recommended that you use cheap SATA disk, rather than expensive fiber channel or SAS disks for ZFS.

Self-healing RAID

This brings us to the single-largest reason why I've become such a ZFS fan. ZFS can detect silent errors, and fix them on the fly. Suppose for a moment that there is bad data on a disk in the array, for whatever reason. When the application requests the data, ZFS constructs the stripe as we just learned, and compares each block against a default checksum in the metadata, which is currently fletcher4. If the read stripe does not match the checksum, ZFS finds the corrupted block, it then reads the parity, and fixes it through combinatorial reconstruction. It then returns good data to the application. This is all accomplished in ZFS itself, without the help of special hardware. Another aspect of the RAIDZ levels is the fact that if the stripe is longer than the disks in the array, if there is a disk failure, not enough data with the parity can reconstruct the data. Thus, ZFS will mirror some of the data in the stripe to prevent this from happening.

Again, if your RAID and filesystem are separate products, they are not aware of each other, so detecting and fixing silent data errors is not possible. So, with that out of the way, let's build some RAIDZ pools. As with my previous post, I'll be using 5 USB thumb drives /dev/sde, /dev/sdf, /dev/sdg, /dev/sdh and /dev/sdi which are all 8 GB in size.

RAIDZ-1

RAIDZ-1 is similar to RAID-5 in that there is a single parity bit distributed across all the disks in the array. The stripe width is variable, and could cover the exact width of disks in the array, fewer disks, or more disks, as evident in the image above. This still allows for one disk failure to maintain data. Two disk failures would result in data loss. A minimum of 3 disks should be used in a RAIDZ-1. The capacity of your storage will be the number of disks in your array times the storage of the smallest disk, minus one disk for parity storage (there is a caveat to zpool storage sizes I'll get to in another post). So in my example, I should have roughly 16 GB of usable disk.

To setup a zpool with RAIDZ-1, we use the "raidz1" VDEV, in this case using only 3 USB drives:

```
# zpool create tank raidz1 sde sdf sdg
# zpool status tank
  pool: pool
 state: ONLINE
 scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        pool          ONLINE       0     0     0
          raidz1-0    ONLINE       0     0     0
            sde       ONLINE       0     0     0
            sdf       ONLINE       0     0     0
            sdg       ONLINE       0     0     0

errors: No known data errors
```

Cleanup before moving on, if following in your terminal:

```
# zpool destroy tank
```

RAIDZ-2

RAIDZ-2 is similar to RAID-6 in that there is a dual parity bit distributed across all the disks in the array. The stripe width is variable, and could cover the exact width of disks in the array, fewer disks, or more disks, as evident in the image above. This still allows for two disk failures to maintain data. Three disk failures would result in data loss. A minimum of 4 disks should be used in a RAIDZ-2. The capacity of your storage will be the number of disks in your array times the storage of the smallest disk, minus two disks for parity storage. So in my example, I should have roughly 16 GB of usable disk.

To setup a zpool with RAIDZ-2, we use the "raidz2" VDEV:

```
# zpool create tank raidz2 sde sdf sdg sdh
# zpool status tank
  pool: pool
 state: ONLINE
 scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        pool          ONLINE       0     0     0
          raidz2-0    ONLINE       0     0     0
            sde       ONLINE       0     0     0
            sdf       ONLINE       0     0     0
            sdg       ONLINE       0     0     0
            sdh       ONLINE       0     0     0

errors: No known data errors
```

Cleanup before moving on, if following in your terminal:

```
# zpool destroy tank
```

RAIDZ-3

RAIDZ-3 does not have a standardized RAID level to compare it to. However, it is the logical continuation of RAIDZ-1 and RAIDZ-2 in that there is a triple parity bit distributed across all the disks in the array. The stripe width is variable, and could cover the exact width of disks in the array, fewer disks, or more disks, as evident in the image above. This still allows for three disk failures to maintain data. Four disk failures would result in data loss. A minimum of 5 disks should be used in a RAIDZ-3. The capacity of your storage will be the number of disks in your array times the storage of the smallest disk, minus three disks for parity storage. So in my example, I should have roughly 16 GB of usable disk.

To setup a zpool with RAIDZ-3, we use the "raidz3" VDEV:

```
# zpool create tank raidz3 sde sdf sdg sdh sdi
# zpool status tank
  pool: pool
 state: ONLINE
 scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        pool          ONLINE       0     0     0
          raidz3-0    ONLINE       0     0     0
            sde       ONLINE       0     0     0
            sdf       ONLINE       0     0     0
            sdg       ONLINE       0     0     0
            sdh       ONLINE       0     0     0
            sdi       ONLINE       0     0     0

errors: No known data errors
```

Cleanup before moving on, if following in your terminal:

```
# zpool destroy tank
```

Hybrid RAIDZ

Unfortunatly, parity-based RAID can be slow, especially when you have many disks in a single stripe (say a 48-disk JBOD). To speed things up a bit, it might not be a bad idea to chop up the single large RAIDZ VDEV into a stripe of multiple RAIDZ VDEVs. This will cost you usable disk space for storage, but can greatly increase performance. Of course, as with the previous RAIDZ VDEVs, the stripe width is variable within each nested RAIDZ VDEV. For each RAIDZ level, you can lose up to that many disks in each VDEV. So, if you

have a stripe of three RAIDZ-1 VDEVs, then you can suffer a total of three disk failures, one disk per VDEV. Usable space would be calculated similarly. In this example, you would lose three disks due to parity storage in each VDEV.

To illustrate this concept, let's suppose we have a 12-disk storage server, and we want to lose as little disk as possible while maximizing performance of the stripe. So, we'll create 4 RAIDZ-1 VDEVs of 3 disks each. This will cost us 4 disks of usable storage, but it will also give us the ability to suffer 4 disk failures, and the stripe across the 4 VDEVs will increase performance.

To setup a zpool with 4 RAIDZ-1 VDEVs, we use the "raidz1" VDEV 4 times in our command. Notice that I've added emphasis on when to type "raidz1" in the command for clarity:

```
# zpool create tank raidz1 sde sdf sdg raidz1 sdh sdi sdj raidz1 sdk sdl sdm raidz1 sdn sdo sdp
# zpool status tank
  pool: pool
 state: ONLINE
 scan: none requested
config:

        NAME          STATE     READ WRITE CKSUM
        pool          ONLINE       0     0     0
          raidz1-0    ONLINE       0     0     0
            sde       ONLINE       0     0     0
            sdf       ONLINE       0     0     0
            sdg       ONLINE       0     0     0
          raidz1-1    ONLINE       0     0     0
            sdh       ONLINE       0     0     0
            sdi       ONLINE       0     0     0
            sdj       ONLINE       0     0     0
          raidz1-2    ONLINE       0     0     0
            sdk       ONLINE       0     0     0
            sdl       ONLINE       0     0     0
            sdm       ONLINE       0     0     0
          raidz1-3    ONLINE       0     0     0
            sdn       ONLINE       0     0     0
            sdo       ONLINE       0     0     0
            sdp       ONLINE       0     0     0

errors: No known data errors
```

Notice now that there are four RAIDZ-1 VDEVs. As mentioned in a previous post, ZFS stripes across VDEVs. So, this setup is essentially a RAIDZ-1+0. Each RAIDZ-1 VDEV will receive 1/4 of the data sent to the pool, then each striped piece will be further striped across the disks in each VDEV. Nested VDEVs can be a great way to keep performance alive and well, long after the pool has been massively fragmented.

Cleanup before moving on, if following in your terminal:

```
# zpool destroy tank
```

Some final thoughts on RAIDZ

Various recommendations exist on when to use RAIDZ-1/2/3 and when not to. Some people say that a RAIDZ-1 and RAIDZ-3 should use an odd number of disks. RAIDZ-1 should start with 3 and not exceed 7 disks in the array, while RAIDZ-3 should start at 7 and not exceed 15. RAIDZ-2 should use an even number of disks, starting with 6 disks and not exceeding 12. This is to ensure that you have an even number of disks the data is actually being written to, and to maximize performance on the array.

~~Instead, in my opinion, you should keep your RAIDZ array at a low power of 2 plus parity. For RAIDZ-1, this is 3, 5 and 9 disks. For RAIDZ-2, this is 4, 6, 10, and 18 disks. For RAIDZ-3, this is 5, 7, 11, and 19 disks.~~ If going north of these recommendations, I would use RAID-1+0 setups personally. This is largely due to the time it will take to rebuild the data (called "resilvering"- a post coming in a bit). Because calculating the parity bit is so expensive, the more disks in the RAIDZ arary, the more expensive this operation will be, as compared to RAID-1+0.

UPDATE: I no longer recommend the "power of 2 plus parity" setup. It's mostly a myth. See http://blog.delphix.com/matt/2014/06/06/zfs-stripe-width/ for a good argument as to why.

Further, I've seen recommendations on the sizes that the disks should be, saying not to exceed 1 TB per disk for RAIDZ-1, 2 TB per disk for RAIDZ-2 and 3 TB per disk for RAIDZ-3. For sizes exceeding these values, you should use 2-way or 3-way mirrors with striping. Whether or not there is any validity to these claims, I cannot say. But, I can tell you that with the fewer number of disks, you should use a RAID level that accomadates your shortcomings. In a 4-disk RAID array, as I have above, calculating multiple parity bits can kill performance. Further, I could suffer at most two disk failures (if using RAID-1+0 or RAIDZ-2). RAIDZ-1 meets somewhere in the middle, where I can suffer a disk failure while stil maintaining a decent level of performance. If I had say 12 disks in the array, then maybe a RAIDZ-1+0 or RAIDZ-3 would be better suited, as the chances of suffering multiple disk failures increases.

Ultimately, you need to understand your storage problem and benchmark your disks. Put them in various RAID configurations, and use a utility such as IOZone 3 to benchmark and stress the array. You know what data you are going to store on the disk. You know what sort of hardware the disks are being installed into. You know what sort of performance you are looking for. It's your decision, and if you spend your time doing research, homework and sleuthing, you will arrive at the right decision. There may be "best practices", but they only go as far as your specific situation.

Lastly, in terms of performance, mirrors will always outperform RAIDZ levels. On both reads and writes. Further, RAIDZ-1 will outperform RAIDZ-2, which it turn will outperform RAIDZ-3. The more parity bits you have to calculate, the longer it's going to take to both read and write the data. Of course, you can always add striping to your VDEVs to maximize on some of this performance. Nested RAID levels, such as RAID-1+0 are considered "the Cadillac of RAID levels" due to the flexibility in which you can lose disks without parity, and the throughput you get from the stripe. So, in a nutshell, from fastest to slowest, your non-nested RAID levels will perform as:

- RAID-0 (fastest)
- RAID-1
- RAIDZ-1
- RAIDZ-2
- RAIDZ-3 (slowest)

Posted by Aaron Toponce on Wednesday, December 5, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 31 } Comments

1. Jon | December 5, 2012 at 9:11 am | Permalink

   Thanks for the pair of articles. I've started messed around with ZFS on one of the scrap server that sits next to my desk. I've read the docs and FAQs, but it's good to see a different perspective of the basic setup.

   I look forward to your next article since, as of last night, one of the drives in the test server has started racking up SMART errors at an alarming rate. I guess I'll get to test resilvering in the real case and not just by faking a drive failure. :O

2. Aaron Toponce | December 5, 2012 at 9:40 am | Permalink

   Np. However, the 3rd post will be covering more VDEVs (there is an order to my chaos). In this case, I'll be covering the L2ARC and the ZIL. Hope to have it up tomorrow morning. Might be a day late though.

3. David | December 5, 2012 at 2:58 pm | Permalink

   Very helpful articles! I've been using ZFS for the past year, and have been extremely impressed by it. Looking forward to your L2ARC and ZIL article, as that's something we'll definitely be wanting to add in the near future.

4. Mark | December 7, 2012 at 11:00 pm | Permalink

   Aaron, I've enjoyed reading the article. Is it really a bad idea to use 5 disks in a RAID-Z2 arrangement? I have 5 x 2TB disks that I want to use in my FreeNAS box, and prefer to have dual parity (rather than RAID-Z1).

5. Aaron Toponce | December 8, 2012 at 7:43 am | Permalink

   "A bad idea", no. However, it's also not optimized. My hypervisors are using RAIDZ-1 with 4 disks, as I needed the space. My motherboard does not have enough SATA ports for 5 disks, and I need more space than what 3 disks would give. Thus, RAIDZ-1 on four disks it is. You do what you can.

6. boneidol | December 28, 2012 at 7:36 pm | Permalink

   "In relatiy" <- trivial typo

7. Aaron Toponce | December 29, 2012 at 6:24 am | Permalink

   Fixed. Thanks!

8. boneidol | December 28, 2012 at 7:42 pm | Permalink

"Instead, in my opinion, you should keep your RAIDZ array at a low power of 2 plus parity. For RAIDZ-1, this is 3, 5 and 9 disks. For RAIDZ-2, this is 4, 8 and 16 disks. For RAIDZ-3, this is 5, 9 and 17 disks"

hi I don't understand these numbers above

Z1 = 2^1 + 1 , 2 ^2 + 1 , 2^3 +1 = 3,5,9
Z2 = 2^1 + 2 , 2^2 +2 , 2^3 +2 = 4,6,10
Z3 = 2^1 + 3 , 2^2 +3 , 2^3 +3 = 5,7,11

Sorry!

9. Alvin | February 2, 2013 at 9:53 pm | Permalink

   Okay here's one for you, I can't find ANY documentation ANYWHERE for using brackets (parentheses) to describe what drives to select when creating a zpool. For example, I am in a VERY sticky situation with money and physical drive constraints. I have figured out a method to make the best use of what I have but it results in a pretty unorthodox (yet completely redundant and failproof [1 drive] way) of getting it all to work AND maximize the use of my motherboard's ports to make it completly expandable in the future. I am basically creating a single-vdev pool containing a bunch of different raid levels, mirrors, and stripes.

   HOWEVER, this is how I have to do it, because of hardware constraints.
   If you were to imagine how to use the zpool create, this is how it would look USING BRACKETS. BUT THERE IS NO MENTION OF HOW TO USE BRACKETS PROPERLY in any zfs documentation. Basically either brackets, commas, &&s, etc, anything that would give me the desired affect.

   zpool create mycoolpool RAIDZ1 ((mirror A B) (mirror C D) (mirror E F) (G) (stripe H, I) (stripe J, K, L) (M))

   Yes I have 7 1TB 'blocks' or 'chunks' in a RAIDZ1, each consisting of different configurations.

   You see, if I were to do this without the brackets, it would create this mess:
   zpool create mycoolpool RAIDZ1 mirror a b mirror c d mirror e f g h i j k l m
   ^^Basically you see here that I would end up with a RAIDZ1 across 3 mirrors, the third of which consisting of a redundancy level such that 8 drives could fail... not what I want.

   And yes, I have indeed seen all the warnings and read countless people say "you shouldn't" but NEVER have I seen anyone deny that it could be done and NEVER have I seen anyone actually answer on HOW to do it.

   I've made up my mind that this is the method and approach that I need to take so please heed your warnings as much as you can as they will be said in vain.

   Thank you very much in advance for a response!!!

10. Aaron Toponce | February 7, 2013 at 10:24 am | Permalink

    No, this is not possible. Other than disks and files, you cannot nest VDEVs. ZFS stripes across RAIDZ and mirror VDEVs, and there's no way around it. You need to rethink your storage.

11. ssl | March 26, 2013 at 11:54 am | Permalink

    I don't quite understand how zfs could recover from certain single disk failures in your example (picture) .. say for example you lost the last drive in your raidz-1 configuration as shown. for the long stripe (A) you lose the parity bit as well as the data in block A4... How could this possibly be recovered, unless zfs puts additional parity blocks in for all stripes whose length exceeds the number of disks??

12. Aaron Toponce | March 27, 2013 at 1:44 pm | Permalink

    Correct. The image isn't 100% accurate. I may fix it, but yes. If you lose too much of a single stripe, then you can't recreate the data. For each stripe written, and this is where my image needs to be updated, a parity bit is written. So, if a stripe crosses the disks twice, then there will be extra parity bits.

    Thanks for pointing this out.

13. Veniamin | April 30, 2013 at 12:54 am | Permalink

    Thanks for articke.
    I wonder how RAIDZ will work with two or more parity stripes.
    I think that in the case of data is longer than recsize x n_data_disks, raidz slpits it into several writes.

14. Aaron Toponce | December 20, 2013 at 11:02 pm | Permalink

I've updated the image (finally) to reflect the inconsistencies I had before.

15. Heny | January 14, 2014 at 9:34 am | Permalink

    ZFS RAIDZ as declustered RAID, how to acheive it?

16. Chris | April 3, 2014 at 2:01 pm | Permalink

    Great articles! Thanks a lot. I was wondering if you have any source for the comments on maximum drive size for the various raidz types? I am very interested why someone thinks maximum 2TB for raidz-2 (as I want to create an array of 8 disks, each 4TB large in a raidz-2 configuration).

17. Aaron Toponce | April 12, 2014 at 3:36 pm | Permalink

    I haven't seen anything regarding maximum drive size. Of course, you need to benchmark your own system, but the more storage you have, the more storage you have. Generally speaking too, the more spindles you have, the better performance will be.

18. TK | July 5, 2014 at 12:21 am | Permalink

    Typo:
    # zpool create tank raidze sde sdf sdg sdh sdi
    should read:
    # zpool create tank raidz3 sde sdf sdg sdh sdi

    That aside, these are all very informative ZFS articles, as are most of your others on th evariety of topics you cover.
    Regards,
    --TK

19. Aaron Toponce | July 5, 2014 at 9:12 am | Permalink

    Fixed! Thanks for the edit.

20. John Naggets | January 30, 2015 at 10:43 am | Permalink

    I am still hesitating about the size of my RAIDZ-2 array. Would it be ok to use 12 disks on a RAIDZ-2 array? isn't that too much? and what about using 9 disks in a RAIDZ-2 array? does the rule of an even number for RAIDZ-2 still apply nowadays?

21. Aaron Toponce | February 17, 2015 at 1:46 pm | Permalink

    > I am still hesitating about the size of my RAIDZ-2 array. Would it be ok to use 12 disks on a RAIDZ-2 array? isn't that too much? and what about using 9 disks in a RAIDZ-2 array? does the rule of an even number for RAIDZ-2 still apply nowadays?

    I wouldn't do RAIDZ2 personally. With 12 disks, I would do RAIDZ1 of 3 disks each. Thus, I would have 4 RAIDZ1 VDEVs:

```
# zpool status pthree
  pool: pthree
 state: ONLINE
  scan: none requested
config:

        NAME             STATE     READ WRITE CKSUM
        pthree           ONLINE       0     0     0
          raidz1-0       ONLINE       0     0     0
            /tmp/file1   ONLINE       0     0     0
            /tmp/file2   ONLINE       0     0     0
            /tmp/file3   ONLINE       0     0     0
          raidz1-1       ONLINE       0     0     0
            /tmp/file4   ONLINE       0     0     0
            /tmp/file5   ONLINE       0     0     0
            /tmp/file6   ONLINE       0     0     0
          raidz1-2       ONLINE       0     0     0
            /tmp/file7   ONLINE       0     0     0
            /tmp/file8   ONLINE       0     0     0
            /tmp/file9   ONLINE       0     0     0
          raidz1-3       ONLINE       0     0     0
            /tmp/file10  ONLINE       0     0     0
            /tmp/file11  ONLINE       0     0     0
            /tmp/file12  ONLINE       0     0     0

errors: No known data errors
```

At least then, you can keep your performance up, while maintaining one disk failure in each VDEV (a total of 3 disk failures maximum). It only comes at the cost of losing 1/3 of the raw disk space, which IMO, isn't that bad.

22. John Naggets | February 18, 2015 at 11:26 am | Permalink

    Thanks for your extensive answer! Actually I was planning to do a RAIDZ-2 of 12 disks because my server can host up to 36 disks. So my plan would be to start with one RAIDZ-2 vdev of 12 disks and then increase the storage always by 12 disks, ending up with 3 RAIDZ-2 vdevs of 12 disks each.

    Or would you still recommend having 12 RAIDZ-1 vdevs of 3 disks each like you mention in your answer? The thing is that with your setup I would be "loosing" in total 12 disks (parity) whereas with my config I would be only "loosing" 6 disks.

23. Aaron Toponce | February 18, 2015 at 1:03 pm | Permalink

    > would you still recommend having 12 RAIDZ-1 vdevs of 3 disks each like you mention in your answer?

    That depends on what you plan on doing with your pool, how much space you need, and how you expect it to perform. If it's just a backup server, that runs nightly backup jobs, and is guaranteed to finish before the next cycle, then performance probably isn't *that* big of a deal (until you need to do a restore at least).

    Regardless, I can't fully answer that. I would build the pool multiple ways, banchmark it, stress test it, fill it and refill it, fail drives, and over all, put it through a stringent series of tests, and see which configuration would be the best for you. Parity-based RAID can be a performance killer but it can be worth it in some scenarios.

24. Ben | May 29, 2015 at 6:14 pm | Permalink

    Thank you so much for putting all this information up. I had spent a good week reading up on ZFS and how to use it, but was still confused beyond belief. I am a long term windows user and I'm just getting into linux and what it can do. Your postings here are laid out so well that it helped me understand how everything works. Thank you again!

25. Rares | June 30, 2015 at 11:52 pm | Permalink

    Amassing documentation. If I'l ever meet you, the beer is on me:D

26. Jim | July 22, 2015 at 11:07 am | Permalink

    Thanks for the ZFS docs. What's the best practice for creating a zpool for use in a RAID1+0 or RAIDZn array from the point of view of future drive replacement?

    What is the likelihood of a replacement drive having a slightly smaller actual capacity than the drive it's replacing? Since we cannot shrink a zpool once created, what would happen if a replacement drive is found to be 1 sector smaller than the failed drive? I assume ZFS issues an error saying that it cannot populate the new drive?

    Is it best practice to manually partition drives prior to adding to the initial zpool so that all members of the pool are a known, precise size? Or is this generally a non-issue?

27. Frank | March 1, 2016 at 7:25 pm | Permalink

    Hi Aaron,
    Thank you once again for your great summaries.
    I'm embarking on building a large array for scientific data storage (actually I'm building two identical arrays, one for backup).
    I wonder if the plan is sane:

    The arrays will require around 100TB of storage eventually but I'm starting with 8x HGST 8TB SAS disks.

    So I was thinking of doing a striped set of two RAIDZ1 vdevs.
    If my calculations are right, this gives 64TB-(2x8TB)=48TB Storage
    The data will be backed up to a clone server using zfs send nightly
    and also to tape.

    NAME
    bigpool
    raidz1-0
    8TB disk1
    8TB disk2
    8TB disk3
    8TB disk4
    raidz1-0

8TB disk1
8TB disk2
8TB disk3
8TB disk4
logs
mirror-1
1.2TB SSD
1.2TB SSD

In due course, I'd add another 8 disks (again as 2x striped RAIDZ1 vdevs)
for a total storage capacity of 96TB (close enough to 100TB).

I'm a little worried that recovering from a disk failure on a vdev with 4x 8TB disks may be a bit risky. The other two options I considered were:
- 8 disk RAIDZ3 array (eventually striped with another 2 of these)
- striped mirrors (but the capacity loss is expensive)

I'd be curious to hear your recommendations

28. Eric | September 10, 2016 at 4:17 am | Permalink

    How come it seems like most documentation say mirrored is always faster than raidz(n), but benchmarks always seem to show the opposite? (https://pthree.org/2012/12/05/zfs-administration-part-ii-raidz/) (https://calomel.org/zfs_raid_speed_capacity.html)

29. gsalerni | January 13, 2017 at 11:08 am | Permalink

    re. Alvins post (9) about trying to assemble a raidZ pool made up of 1tb vdevs which were in turn a variety of single disks, mirrors and stripes. Although you can't nest vdevs (other than disks and files) - could he not use madam to construct the various 1tb metadisks using md mirrors and stripes as required and then create a zfs raidz out of those? I imagine that wouldn't perform great but would it work? zfs wouldn't care that the raw disks were in fact meta disks would it?

30. TMS | March 5, 2017 at 10:47 am | Permalink

    Very nice article, but you are incorrect wehn you say mirror is ALWAYS faster. No it isn't. For sequential reads Raidz is faster. Same with writes. IOPS are always faster on a mirror.

31. xaoc | August 22, 2017 at 3:04 am | Permalink

    I have strange situation and can't explain it . I will appreciate your comment on bellow setup:
    zpool list
    NAME SIZE ALLOC FREE EXPANDSZ FRAG CAP DEDUP HEALTH ALTROOT
    test_3x3s 327T 1.11M 327T - 0% 0% 1.00x ONLINE -
    dmadm@s1349014530:~$ sudo zpool status
    pool: test_3x3s
    state: ONLINE
    scan: none requested
    config:

    NAME STATE READ WRITE CKSUM
    test_3x3s ONLINE 0 0 0
    raidz3-0 ONLINE 0 0 0
    sdc ONLINE 0 0 0
    sdd ONLINE 0 0 0
    sde ONLINE 0 0 0
    sdf ONLINE 0 0 0
    sdg ONLINE 0 0 0
    sdh ONLINE 0 0 0
    sdi ONLINE 0 0 0
    sdj ONLINE 0 0 0
    sdk ONLINE 0 0 0
    sdl ONLINE 0 0 0
    sdm ONLINE 0 0 0
    sdn ONLINE 0 0 0
    raidz3-1 ONLINE 0 0 0
    sdo ONLINE 0 0 0
    sdp ONLINE 0 0 0
    sdq ONLINE 0 0 0

sdr ONLINE 0 0 0
sds ONLINE 0 0 0
sdt ONLINE 0 0 0
sdu ONLINE 0 0 0
sdv ONLINE 0 0 0
sdw ONLINE 0 0 0
sdx ONLINE 0 0 0
sdy ONLINE 0 0 0
sdz ONLINE 0 0 0
raidz3-2 ONLINE 0 0 0
sdaa ONLINE 0 0 0
sdab ONLINE 0 0 0
sdac ONLINE 0 0 0
sdad ONLINE 0 0 0
sdae ONLINE 0 0 0
sdaf ONLINE 0 0 0
sdag ONLINE 0 0 0
sdah ONLINE 0 0 0
sdai ONLINE 0 0 0
sdaj ONLINE 0 0 0
sdak ONLINE 0 0 0
sdal ONLINE 0 0 0

errors: No known data errors
df -h
Filesystem Size Used Avail Use% Mounted on
udev 189G 0 189G 0% /dev
tmpfs 38G 850M 37G 3% /run
/dev/md0 103G 1.9G 96G 2% /
tmpfs 189G 0 189G 0% /dev/shm
tmpfs 5.0M 0 5.0M 0% /run/lock
tmpfs 189G 0 189G 0% /sys/fs/cgroup
tmpfs 38G 0 38G 0% /run/user/1002
test_3x3s 231T 256K 231T 1% /test_3x3s
##
zpool list
NAME SIZE ALLOC FREE EXPANDSZ FRAG CAP DEDUP HEALTH ALTROOT
test_3x3s 326T 1.11M 326T - 0% 0% 1.00x ONLINE -
dmadm@s1349014530:~$ df -h
Filesystem Size Used Avail Use% Mounted on
udev 189G 0 189G 0% /dev
tmpfs 38G 858M 37G 3% /run
/dev/md0 103G 1.9G 96G 2% /
tmpfs 189G 0 189G 0% /dev/shm
tmpfs 5.0M 0 5.0M 0% /run/lock
tmpfs 189G 0 189G 0% /sys/fs/cgroup
tmpfs 38G 0 38G 0% /run/user/1002
test_3x3s 230T 256K 230T 1% /test_3x3s
zpool status
pool: test_3x3s
state: ONLINE
scan: none requested
config:

NAME STATE READ WRITE CKSUM
test_3x3s ONLINE 0 0 0
raidz3-0 ONLINE 0 0 0
sdc ONLINE 0 0 0
sdd ONLINE 0 0 0
sde ONLINE 0 0 0
sdf ONLINE 0 0 0
sdg ONLINE 0 0 0
sdh ONLINE 0 0 0
sdi ONLINE 0 0 0
sdj ONLINE 0 0 0
sdk ONLINE 0 0 0

sdl ONLINE 0 0 0
sdm ONLINE 0 0 0
sdn ONLINE 0 0 0
sdo ONLINE 0 0 0
sdp ONLINE 0 0 0
sdq ONLINE 0 0 0
sdr ONLINE 0 0 0
sds ONLINE 0 0 0
sdt ONLINE 0 0 0
raidz3-1 ONLINE 0 0 0
sdu ONLINE 0 0 0
sdv ONLINE 0 0 0
sdw ONLINE 0 0 0
sdx ONLINE 0 0 0
sdy ONLINE 0 0 0
sdz ONLINE 0 0 0
sdaa ONLINE 0 0 0
sdab ONLINE 0 0 0
sdac ONLINE 0 0 0
sdad ONLINE 0 0 0
sdae ONLINE 0 0 0
sdaf ONLINE 0 0 0
sdag ONLINE 0 0 0
sdah ONLINE 0 0 0
sdai ONLINE 0 0 0
sdaj ONLINE 0 0 0
sdak ONLINE 0 0 0
sdal ONLINE 0 0 0

In few words ... If I undesrtand it correctly:
2 VDEVs RAIDZ3 should use 6 disks for parity (3 for each VDEV)
3 VDEVs RAIDZ3 should use 9 disks for parity (3 for each VDEV)
And it is logical to have less usable space with 3 VDEVs compared with 2 VDEVs, but practicaly it seems that with 2 VDEVs configuration I have less usable space?

{ 13 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | December 6, 2012 at 6:00 am | Permalink

   [...] The previous post about using ZFS with GNU/Linux concerned covering the three RAIDZ virtual devices .... This post will cover another VDEV- the ZFS Intent Log, or the ZIL. [...]

2. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 13, 2012 at 5:59 am | Permalink

   [...] RAIDZ [...]

3. Aaron Toponce : Install ZFS on Debian GNU/Linux | December 13, 2012 at 6:05 am | Permalink

   [...] RAIDZ [...]

4. Aaron Toponce : ZFS Administration, Part VI- Scrub and Resilver | December 13, 2012 at 6:07 am | Permalink

   [...] RAIDZ [...]

5. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | December 20, 2012 at 8:06 am | Permalink

   [...] RAIDZ [...]

6. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | December 20, 2012 at 8:07 am | Permalink

   [...] RAIDZ [...]

7. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | January 7, 2013 at 9:23 pm | Permalink

   [...] RAIDZ [...]

8. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | January 7, 2013 at 9:25 pm | Permalink

[...] RAIDZ [...]

9. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | February 20, 2013 at 8:33 am | Permalink

   [...] RAIDZ [...]

10. Aaron Toponce : ZFS Administration, Part IX- Copy-on-write | April 19, 2013 at 4:57 am | Permalink

    [...] RAIDZ [...]

11. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:03 am | Permalink

    [...] RAIDZ [...]

12. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | July 2, 2013 at 7:24 am | Permalink

    […] RAIDZ […]

13. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | July 8, 2013 at 10:08 pm | Permalink

    […] RAIDZ […]

Aaron Toponce

{ 2012.12.06 }

ZFS Administration, Part III- The ZFS Intent Log

Table of Contents

The previous post about using ZFS with GNU/Linux concerned covering the three RAIDZ virtual devices (VDEVs). This post will cover another VDEV- the ZFS Intent Log, or the ZIL.

Terminology

Before we can begin, we need to get a few terms out of the way that seem to be confusing people on forums, blog posts, mailing lists, and general discussion. It confused me, even though I understood the end goal, right up to the writing of this post. So, let's get at it:

- ZFS Intent Log, or ZIL- A logging mechanism where all of the data to be the written is stored, then later flushed as a transactional write. Similar in function to a journal for journaled filesystems, like ext3 or ext4. Typically stored on platter disk. Consists of a ZIL header, which points to a list of records, ZIL blocks and a ZIL trailer. The ZIL behaves differently for different writes. For writes smaller than 64KB (by default), the ZIL stores the write data. For writes larger, the write is not stored in the ZIL, and the ZIL maintains pointers to the synched data that is stored in the log record.
- Separate Intent Log, or SLOG- A separate logging device that caches the synchronous parts of the ZIL before flushing them to slower disk. This would either be a battery-backed DRAM drive or a fast SSD. The SLOG only caches synchronous data, and does not cache asynchronous data. Asynchronous data will flush directly to spinning disk. Further, blocks are written a block-at-a-time, rather than as simultaneous transactions to the SLOG. If the SLOG exists, the ZIL will be moved to it rather than residing on platter disk. Everything in the SLOG will always be in system memory.

When you read online about people referring to "adding an SSD ZIL to the pool", they are meaning adding an SSD SLOG, of where the ZIL will reside. The ZIL is a subset of the SLOG in this case. The SLOG is the device, the ZIL is data on the device. Further, not all applications take advantage of the ZIL. Applications such as databases (MySQL, PostgreSQL, Oracle), NFS and iSCSI targets do use the ZIL. Typical copying of data around the filesystem will not use it. Lastly, the ZIL is generally never read, except at boot to see if there is a missing transaction. The ZIL is basically "write-only", and is very write-intensive.

SLOG Devices

Which device is best for a SLOG? In order from fastest to slowest, here's my opinion:

1. NVRAM- A battery-backed DRAM drive, such as the ZeusRAM SSD by STEC. Fastest and most reliable of this list. Also most expensive.
2. SSDs- NAND flash-based chips with wear-leveling algorithms. Something like the PCI-Express OCZ SSDs or Intel. Preferably should be SLC, although the gap between SLC and MLC SSDs is thinning.
3. 10k+ SAS drives- Enterprise grade, spinning platter disks. SAS and fiber channel drives push IOPS over throughput, typically twice as fast as consumer-grade SATA. Slowest and least reliable of this list. Also the cheapest.

It's important to identify that all three devices listed above can maintain data persistence during a power outage. The SLOG and the ZIL are critical in getting your data to spinning platter. If a power outage occurs, and you have a volatile SLOG, the worst thing that will happen is the new data is not flushed, and you are left with old data. However, it's important to note, that in the case of a power outage, you won't have corrupted data, just lost data. Your data will still be consistent on disk.

SLOG Performance

Because the SLOG is fast disk, what sort of performance can I expect to see out of my application or system? Well, you will see improved disk latencies, disk utilization and system load. What you won't see is improved throughput. Remember that the SLOG device is still flushing data to platter every 5 seconds. As a result, benchmarking disk after adding a SLOG device doesn't make much sense, unless the goal of the benchmark is to test synchronous disk write latencies. So, I don't have those numbers for you to gravel over. What I do have, however, are a few graphs.

I have a virtual machine that is disk-write intensive. It is a disk image on a GlusterFS replicated filesystem on a ZFS dataset. I have plenty of RAM in the hypervisor, a speedy CPU, but slow SATA disk. Due to the applications on this virtual machine wanting to write many graphs to disks frequently, with the

graphs growing, I was seeing ~5-10 second disk latencies. The throughput on the disk was intense. As a result, doing any writes in the VM was painful. System upgrades, modifying configuration files, even logging in. It was all very, very slow.

So, I partitioned my SSDs, and added the SLOG. Immediately, my disk latencies dropped to around 200 milliseconds. Disk utilization dropped from around 50% busy to around 5% busy. System load dropped from 1-2 to almost non-existent. Everything concerning the disks is in a much more healthy state, and the VM is much more happy. As proof, look at the following graphs preserved here from http://zen.ae7.st/munin/.

This first image shows my disks from the hypervisor perspective. Notice that the throughput for each device was around 800 KBps. After adding the SSD SLOG, the throughput dropped to 400 KBps. This means that the underlying disks in the pool are doing less work, and as a result, will last longer.

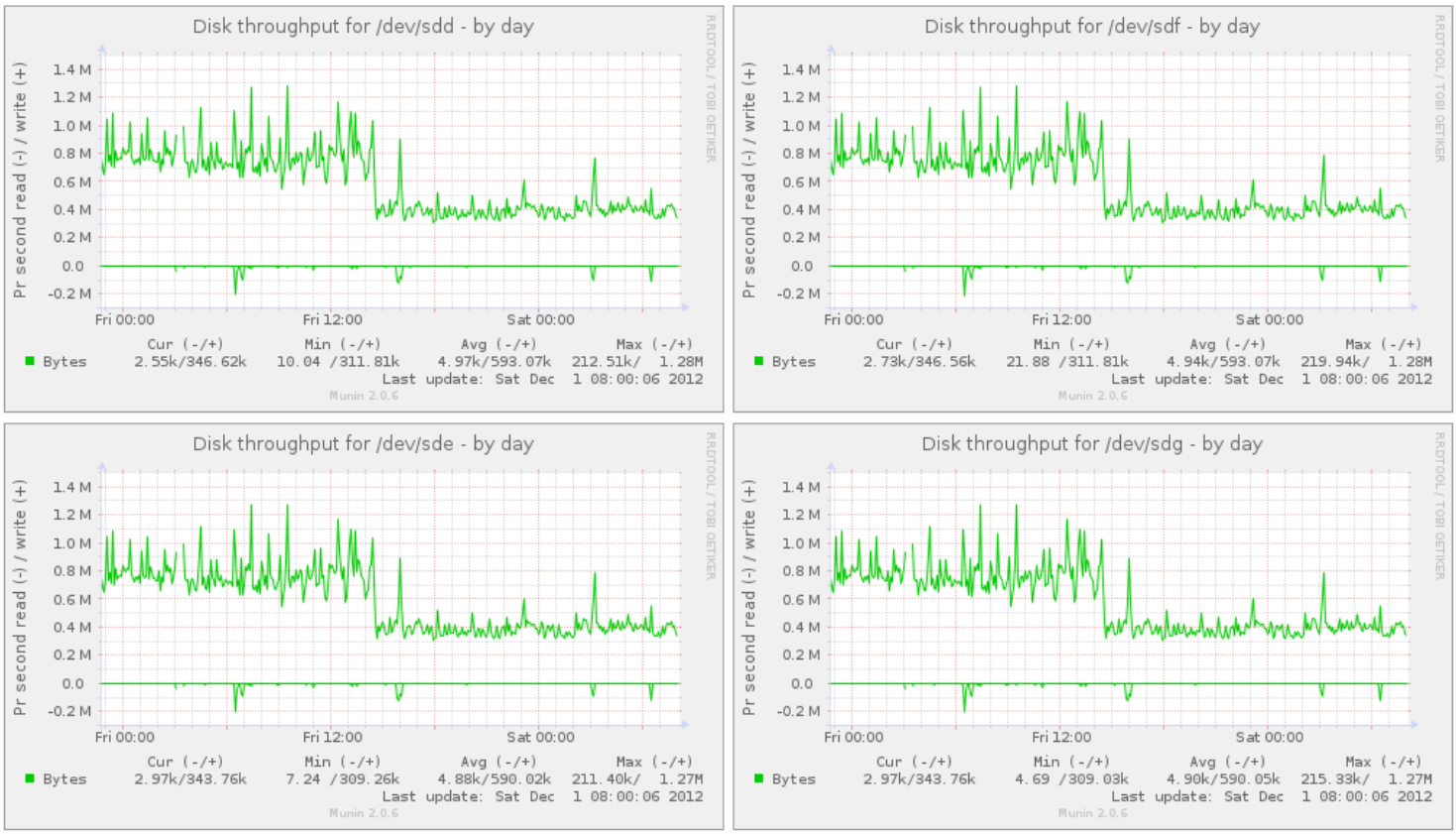


Image showing all 4 disks throughput in the zpool on the hypervisor.

This next image show my disk from the virtual machine perspective. Notice how disk latency and utilization drop as explained above, including system load.

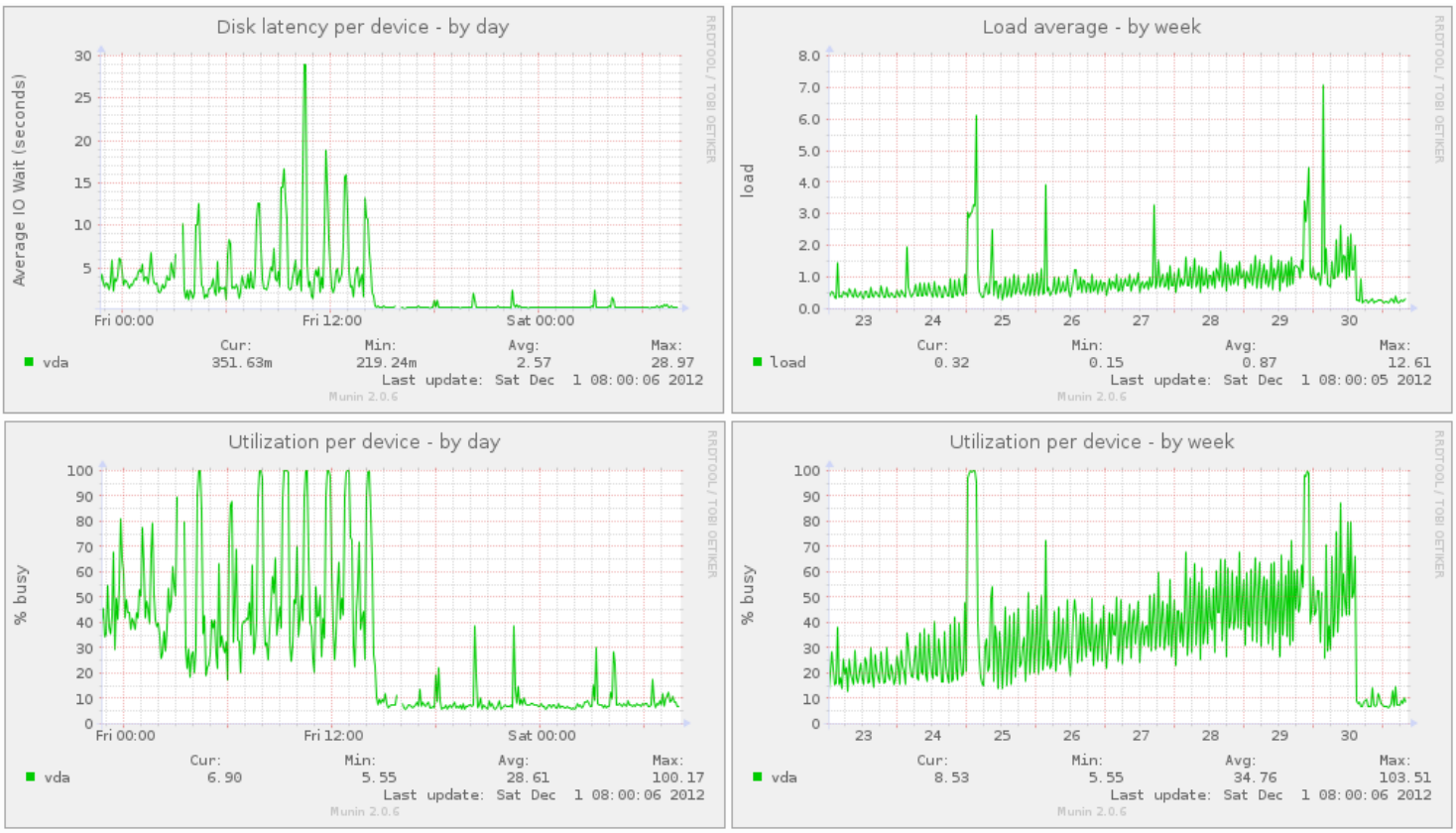


Image showing what sort of load the virtual machine was under.

I blogged about this just a few days ago at http://pthree.org/2012/12/03/how-a-zil-improves-disk-latencies/.

Adding a SLOG

WARNING: Some motherboards will not present disks in a consistent manner to the Linux kernel across reboots. As such, a disk identified as /dev/sda on one boot might be /dev/sdb on the next. For the main pool where your data is stored, this is not a problem as ZFS can reconstruct the VDEVs based on the metadata geometry. For your L2ARC and SLOG devices, however, no such metadata exists. So, rather than adding them to the pool by their /dev/sd? names, you should use the /dev/disk/by-id/* names, as these are symbolic pointers to the ever-changing /dev/sd? files. If you don't heed this warning, your SLOG device may not be added to your hybrid pool at all, and you will need to re-add it later. This could drastically affect the performance of the applications depending on the existence of a fast SLOG.

Adding a SLOG to your existing zpool is not difficult. However, it is considered best practice to mirror the SLOG. So, I'll follow best practice in this example. Suppose I have 4 platter disks in my pool, and an OCZ Revodrive SSD that presents two 60 GB drives to the system. I'll partition the drives on the SSD, for 5 GB, then mirror the partitions as my SLOG. This is how you would add the SLOG to the pool. Here, I am using GNU parted to create the partitions first, then adding the SSDs. The devices in /dev/disk/by-id/ are pointing to /dev/sda and /dev/sdb. FYI.

```
# parted /dev/sda mklabel gpt mkpart primary zfs 0 5G
# parted /dev/sdb mklabel gpt mkpart primary zfs 0 5G
# zpool add tank log mirror \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1 \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1
# zpool status
  pool: tank
 state: ONLINE
 scan: scrub repaired 0 in 1h8m with 0 errors on Sun Dec  2 01:08:26 2012
config:

        NAME                                            STATE     READ WRITE CKSUM
        pool                                            ONLINE       0     0     0
          raidz1-0                                      ONLINE       0     0     0
            sdd                                         ONLINE       0     0     0
            sde                                         ONLINE       0     0     0
            sdf                                         ONLINE       0     0     0
            sdg                                         ONLINE       0     0     0
        logs
          mirror-1                                      ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1  ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1  ONLINE       0     0     0
```

SLOG Life Expectancy

Because you will likely be using a consumer-grade SSD for your SLOG in your GNU/Linux server, we need to make some mention of the wear and tear of SSDs for write-intensive scenarios. Of course, this will largely vary based on manufacturer, but we can setup some generalities.

First and foremost, ZFS has advanced wear-leveling algorithms that will evenly wear each chip on the SSD. There is no need for TRIM support, which in all reality, is really just a garbage collection support more than anything. The wear-leveling of ZFS in inherent due to the copy-on-write nature of the filesystem.

Second, various drives will be implemented with different nanometer processes. The smaller the nanometer process, the shorter the life of your SSD. As an example, the Intel 320 is a 25 nanometer MLC 300 GB SSD, and is rated at roughly 5000 P/E cycles. This means you can write to your entire SSD 5000 times if using wear leveling algorithms. This produces 1500000 GB of total written data, or 1500 TB. My ZIL maintains about 3 MB of data per second. As a result, I can maintain about 95 TB of written data per year. This gives me a life of about 15 years for this Intel SSD.

However, the Intel 335 is a 20 nanometer MLC 240 GB SSD, and is rated at roughly 3000 P/E cycles. With wear leveling, this means you can write you entire SSD 3000 times, which produces 720 TB of total written data. This is only 7 years for my 3 MBps ZIL, which is less than 1/2 the life expectancy the Intel 320. Point is, you need to keep an eye on these things when planning out your pool.

Now, if you are using a battery-backed DRAM drive, then wear leveling is not a problem, and the DIMMs will likely last the duration of your server. Same might be said for 10k+ SAS or FC drives.

Capacity

Just a short note to say that you will likely not need a large ZIL. I've partitioned my ZIL with only 4 GB of usable space, and it's barely occupying a MB or two of space. I've put all my virtual machines on the same hypervisor, ran operating system upgrades, while they were also doing a great amount of work, and only saw the ZIL get up to about 100 MB of cached data. I can't imagine what sort of workload you would need to get your ZIL north of 1 GB of used space, let alone the 4 GB I partitioned off. Here's a command you can run to check the size of your ZIL:

```
# zpool iostat -v tank
                                                capacity     operations    bandwidth
tank                                          alloc   free   read  write   read  write
--------------------------------------------  -----  -----  -----  -----  -----  -----
tank                                           839G  2.81T     76      0  1.86M      0
  raidz1                                       839G  2.81T     73      0  1.81M      0
    sdd                                           -      -     52      0   623K      0
    sde                                           -      -     47      0   620K      0
    sdf                                           -      -     50      0   623K      0
    sdg                                           -      -     47      0   620K      0
logs                                              -      -      -      -      -      -
  mirror                                      1.46M  3.72G     20      0   285K      0
    ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1      -      -     20      0   285K      0
    ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1      -      -     20      0   285K      0
--------------------------------------------  -----  -----  -----  -----  -----  -----
```

Conclusion

A fast SLOG can provide amazing benefits for applications that need lower latencies on synchronous transactions. This works well for database servers or other applications that are more time sensitive. However, there is increased cost for adding a SLOG to your pool. The battery-backed DRAM chips are very, very expensive. Usually on the order of $2,500 per 8 GB of DDR3 DIMMs, where a 40 GB MLC SSD can cost you only $100, and a 600 GB 15k SAS drive is $200. Again though, capacity really isn't an issue, while performance is. I would go for faster IOPS on the SSD, and a smaller capacity. Unless you want to partition it, and share the L2ARC on the same drive, which is a great idea, and something I'll cover in the next post.

Posted by Aaron Toponce on Thursday, December 6, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 22 } Comments

1. David E. Anderson | December 6, 2012 at 6:59 am | Permalink

   Excellent series, Aaron! Very helpful...

2. Michael | March 20, 2013 at 9:09 am | Permalink

   Aaron, I really appreciate your series. Condenses and drives home the volumes of stuff read on ZFS in plain English with examples. One question... What did you use to track/graph the disk usage over time ?

3. Michael | March 20, 2013 at 9:11 am | Permalink

   Note regarding my previous post. I am using RHEL 6 but still finding much benefit from your posts. I would like to track and graph the iowaits and such over time like you did.

4. Aaron Toponce | March 20, 2013 at 9:50 am | Permalink

   Those graphs are from Munin. http://munin-monitoring.org/

5. Michael | March 20, 2013 at 2:14 pm | Permalink

Thanks for the series and the quick response with the tip. I tried the URL "http://zen.ae7.st/munin/" in the article with no response but the URL in the comment worked fine & now I am researching "http://munin-monitoring.org/"

6. Aaron Toponce | March 20, 2013 at 2:37 pm | Permalink

   Yeah. I took it down, because it was thrashing my drives due to 250 MB snapshot differentials every 15 minutes. I need a better solution.

7. Alan | June 24, 2013 at 7:14 pm | Permalink

   No. TRIM *is* needed, even under ZFS. It's TRIM that tells the drive what blocks are actually free, as the drive has no knowledge of what is or is not allocated/in use by the filesystem. This is unrelated to ZFS's COW operation. Many SSDs use NAND memory, which must be zeroed before data is stored. Without TRIM, the SSD won't know when the filesystem (ZFS or other) is no longer using a block, so it will have to zero out a block each time it's written to. With TRIM, the drive is aware of what blocks aren't in use, and can zero them out ahead of time instead of at the actual time of write. This is the "garbage collection" SSDs perform, and it is unrelated to any "garbage collection" that might occur at the filesystem level. Wear leveling is also important, but that's also a function of the drive and not the filesystem. Since logical blocks on a drive do not necessarily map to specific physical blocks on modern drives (e.g., spare blocks can be mapped in by the drive in a way transparent to the OS and user), "wear leveling" (i.e., trying to distribute writes physically across a device) in the filesystem is futile at best.

8. Warren Downs | July 1, 2013 at 12:01 pm | Permalink

   Correct me if I'm wrong, but if you "will likely not need a large ZIL" and decide to only use a few GBs for it, wouldn't you need to calculate your life expectancy using only the small portion of disk that will be repeatedly written to SSD? This would produce a much shorter life expectancy. For your example, if 5 GB is used for 5000 write cycles you'd have only 25Tb, or less than a third of a year life expectancy.

   Am I missing something?

9. Aaron Toponce | July 1, 2013 at 12:31 pm | Permalink

   Use an SSD that has wear leveling algorithms built into the driver, then ZFS can take advantage of them, and you won't chew through the same physical chips on the board.

10. Warren Downs | July 1, 2013 at 3:23 pm | Permalink

    Makes sense-- but then even though "ZFS has advanced wear-leveling algorithms" you won't be using the ZFS feature. In other words, if you want to go cheap and depend on ZFS wear leveling, you'd need to give it the whole device to work with, not just a small portion of it.

11. Aaron Toponce | July 2, 2013 at 7:17 am | Permalink

    No, you don't. Again, ZFS will wear the SSD correctly. The partition will move across the chips evenly, and every chip will get the same amount of wear as the rest.

12. Warren Downs | July 2, 2013 at 11:30 am | Permalink

    Not to belabor this point unnecessarily, but that can only be true if:

    1. The SSD device supports wear leveling, or
    2. ZFS has access to the whole device, so it can do the wear leveling.

    My point is: Don't try to save money by using SSD devices that don't support wear leveling, if you aren't planning to give ZFS access to the whole device.

13. Ahmed Kamal | July 21, 2013 at 5:10 pm | Permalink

    Hi Aaron,

    This series is awesome! It wasn't clear to me exactly how does SLOG device help performance that much! I mean, the data still lives in RAM and has to eventually be written to spinning rust. My wild guess is that the performance gains are from quickly ACK'ing synchronous writes, and mostly converting those mostly random writes to mostly streaming writes, reducing seeking on mechanical disk heads. However I can't be sure, looking for a confirmation from you

14. Aaron Toponce | August 7, 2013 at 10:03 am | Permalink

    The performance gains come from the low latency of the SSD. The synchronous write is written to the SSD, which acks back quickly. Then the data is flushed from RAM to spinning platter, as you mention. So, the performance increase comes from the SSD's low latencies.

15. Dzezik | September 12, 2013 at 3:01 pm | Permalink

    to Alan. TRIM and GC is not the same, and WL is another one. TRIM can force to reorganize some cells to clean whole blocks after OS deletes some data from drive. GC will do the same but not on OS command but in background. dont bother TRIM for new SSD with advanced GC. old SSD has poor GC and TRIM was only one way to prepare new blocks for newwritten data.

16. Alberto | October 5, 2013 at 11:42 am | Permalink

    I got a bit lost in here:
    "It is a disk image on a GlusterFS replicated filesystem on a ZFS dataset."

    Does this mean that the images (and their virtual hard-drives) are actually and physically "over" a GlusterFS, and then, over it you use ZFS??

If so..., is the ZIL disk directly connected to the virtual machine?? Or through GlusterFS as well?

17. Mark | March 12, 2014 at 4:31 am | Permalink

    "First and foremost, ZFS has advanced wear-leveling algorithms that will evenly wear each chip on the SSD."

    If I have 240Gb SSD, and create two partitions 10GB (SLOG), 230GB (ARC). Under a heavy write scenario (ie migrating 12TB of data onto the pool), doesn't that mean that the SLOG wear will only be on 10GB section of the SSD? Or does zfs some how know the drive ssd and remap the writes over the enitre ssd

    http://www.slideshare.net/dpavlin/cuc2013-zfs (page 6, last point) seems to indicate if you need 10GB of SLOG, you should create a partition of 100GB. Would you agree with this?

    To minimize the wear on the ssd, is it advisable to create a ram drive for the slog during migration and then when the migration is finished, switch over to the ssd? The point being that during the migration the server is baby sat and the migration can be restarted if there is a power outage etc

18. Anakha | April 28, 2014 at 3:36 am | Permalink

    Dzezik - That is not quite how it works. How does the GC algorithm know which blocks are empty? Without TRIM (or knowledge of the filesystem in use) the only way is when a block is overwritten. GC uses TRIM to know which logical blocks were written but are now considered empty (deleted on the FS) before they're overwritten. This also plays into wear leveling because GC moves around much more invalid (written once but not marked by TRIM as deleted by the OS) data for writes that don't fill a physical "block" (a page).

    These considerations along with the fact that logical blocks aren't statically mapped to physical pages (it's a dynamic mapping maintained by the SSD firmware internally on excess flash that is not user accessible) are why the author's statements on not needing TRIM and ZFS providing wear leveling aren't accurate. ZFS can not provide this function for the drive.

19. Sergiy Mashtaler | May 17, 2016 at 2:20 pm | Permalink

    Just a "little" tweak to make actually your ZIL to do the work you have to set sync to "enabled" otherwise it is not used and basically is a waist. I couldn't understand why alloc on my log was 0.
    To enable sync execute zfs set sync=always zpoolName/poolSet

20. Attila | September 29, 2016 at 1:10 pm | Permalink

    Hello,

    I'd like to know what happens when - instead of a power outage - the SSD containing the SLOG breaks.
    It's clear that the cached-to-be-written data is lost. Is there anything different from the power-loss scenario? Anything else than pool is "degraded"? Needed metadata missing?

    Thanks in advance!

21. Nawang Lama | September 26, 2017 at 10:12 am | Permalink

    Hi Aaron,
    We are looking for some kind performance tunning in ZFS. So will you be able to help us to do so. If yes please mail me at nawang81@gmail.com or share me your email address.

22. Michel Erb | May 30, 2018 at 11:08 am | Permalink

    To confirm an assumption, if this statement is true "ZFS will wear the SSD correctly. The partition will move across the chips evenly, and every chip will get the same amount of wear as the rest.", that means a larger disk, with more chips, takes more time to wear out or the smallest disk, is not always the best option considering longevity.

{ 7 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | December 7, 2012 at 6:00 am | Permalink

    [...] Our continuation in the ZFS Administration series continues with another zpool VDEV call the Adjustable Replacement Cache, or ARC for short. Our previous post discussed the ZFS Intent Log, or ZIL and the Separate Intent Logging Device, or S.... [...]

2. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 13, 2012 at 6:06 am | Permalink

    [...] The ZFS Intent Log (ZIL) [...]

3. Aaron Toponce : Install ZFS on Debian GNU/Linux | December 18, 2012 at 12:33 pm | Permalink

    [...] The ZFS Intent Log (ZIL) [...]

4. ZFS administration (3) « 0ddn1x: tricks with *nix | December 26, 2012 at 10:16 am | Permalink

    [...] http://pthree.org/2012/12/06/zfs-administration-part-iii-the-zfs-intent-log/ [...]

5. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 29, 2012 at 6:22 am | Permalink

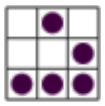
    [...] The ZFS Intent Log (ZIL) [...]

6. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | January 7, 2013 at 9:24 pm | Permalink

   [...] The ZFS Intent Log (ZIL) [...]

7. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 8:15 am | Permalink

   [...] The ZFS Intent Log (ZIL) [...]



Aaron Toponce

{ 2012.12.07 }

ZFS Administration, Part IV- The Adjustable Replacement Cache

Table of Contents

Our continuation in the ZFS Administration series continues with another zpool VDEV call the Adjustable Replacement Cache, or ARC for short. Our previous post discussed the ZFS Intent Log, or ZIL and the Separate Intent Logging Device, or SLOG.

Traditional Caches

Caching mechanisms on Linux and other operating systems use what is called a Least Recently Used caching algorithm. The way the LRU algorithm works, is when an application reads data blocks, they are put into the cache. The cache will fill as more and more data is read, and put into the cache. However, the cache is a FIFO (first in, first out) algorithm. Thus, when the cache is full, the older pages will be pushed out of the cache. Even if those older pages are accessed more frequently. Think of the whole process as a conveyor belt. Blocks are put into most recently used portion of the cache. As more blocks are read, the push the older blocks toward the least recently used portion of the cache, until they fall off the conveyor belt, or in other words are evicted.

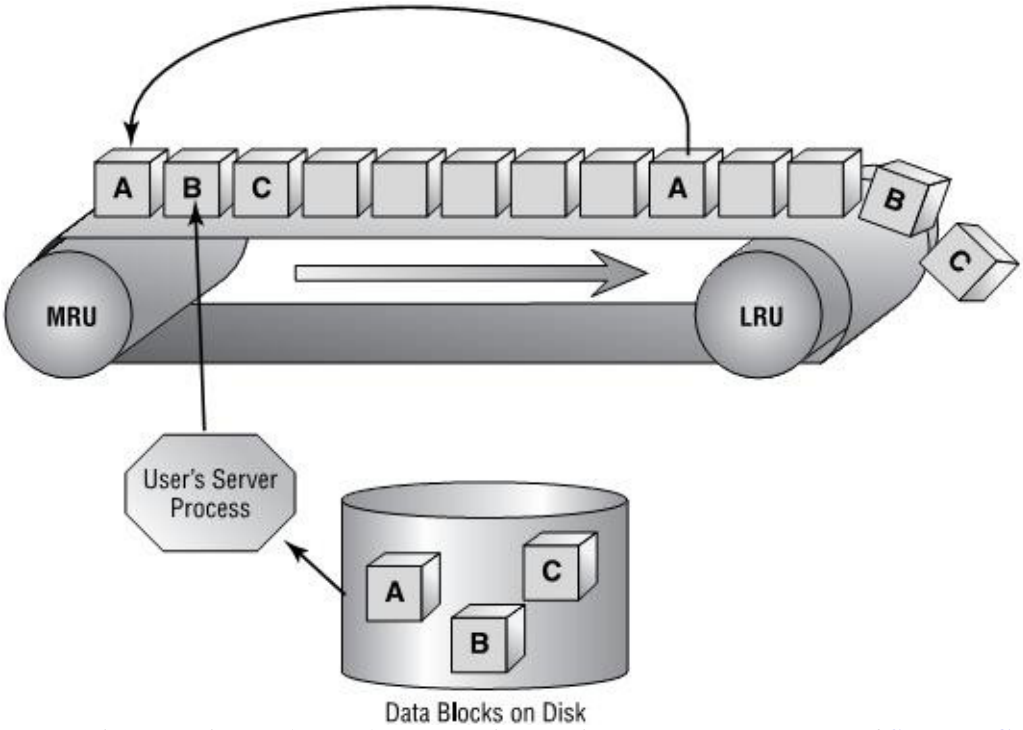


Image showing the traditional LRU caching scheme. Image courtesy of Storage Gaga.

When large sequential reads are read from disk, and placed into the cache, it has a tendency to evict more frequently requested pages from the cache. Even if this data was only needed once. Thus, from the cache perspective, it ends up with a lot of worthless, useless data that is no longer needed. Of course, it's eventually replaced as newer data blocks are requested.

There do also exist least frequently used (LFU) caches. However, they suffer from the problem that newer data could be evicted from the cache if it's not read frequently enough. Thus, there are a great amount of disk requests, which kind of defeats the purpose of a cache to begin with. So, it would seem that the obvious approach would be to somehow combine the two- have an LRU and an LFU simultaneously.

The ZFS ARC

The ZFS adjustable replacement cache (ARC) is one such caching mechanism that caches both recent block requests as well as frequent block requests. It is an implementation of the patented IBM adaptive replacement cache, with some modifications and extensions.

Before beginning, I should mention that I learned a lot about the ZFS ARC from http://www.c0t0d0s0.org/archives/5329-Some-insight-into-the-read-cache-of-ZFS-or-The-ARC.html. The following images of that post I am reusing here. Thank you Joerg for an excellent post.

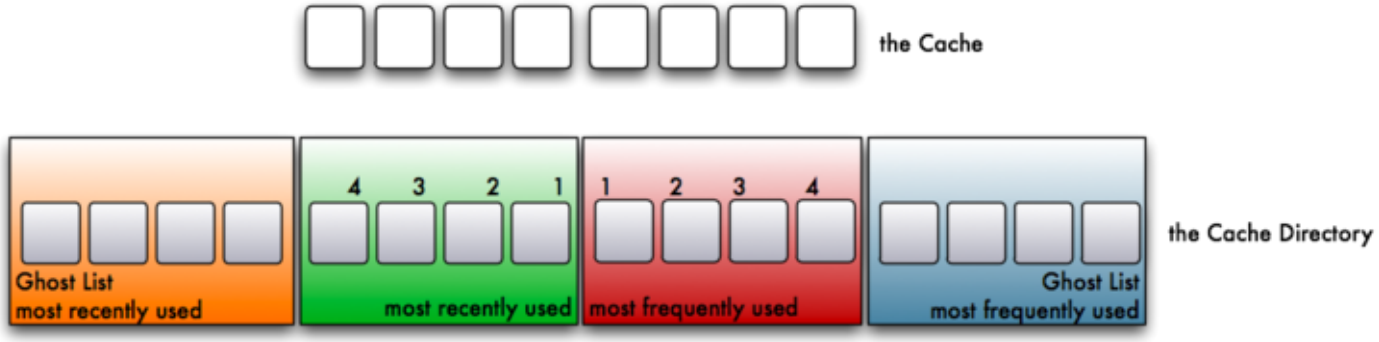
Terminology

- Adjustable Replacement Cache, or ARC- A cache residing in physical RAM. It is built using two caches- the most frequently used cached and the most recently used cache. A cache directory indexes pointers to the caches, including pointers to disk called the ghost frequently used cache, and the ghost most recently used cache.
- Cache Directory- An indexed directory of pointers making up the MRU, MFU, ghost MRU and ghost MFU caches.
- MRU Cache- The most recently used cache of the ARC. The most recently requested blocks from the filesystem are cached here.
- MFU Cache- The most frequently used cache of the ARC. The most frequently requested blocks from the filesystem are cached here.

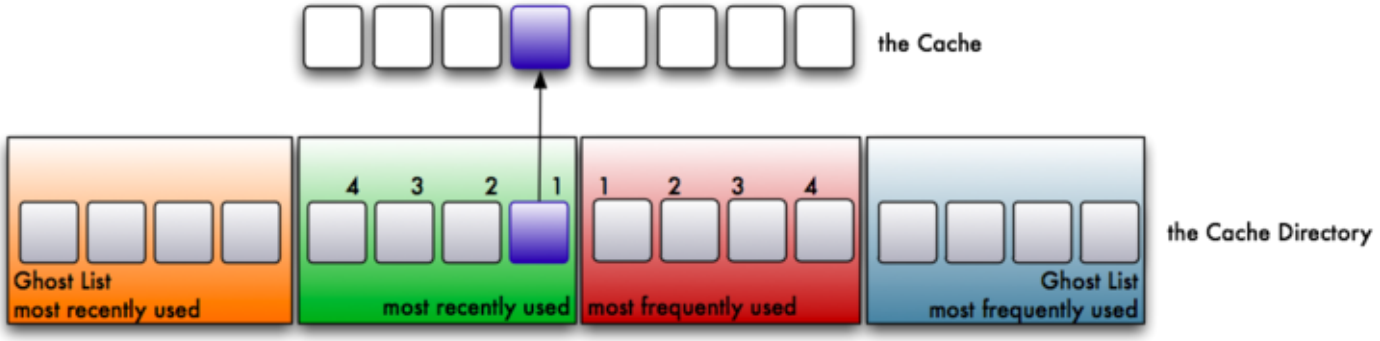
- Ghost MRU- Evicted pages from the MRU cache back to disk to save space in the MRU. Pointers still track the location of the evicted pages on disk.
- Ghost MFU- Evicted pages from the MFU cache back to disk to save space in the MFU. Pointers still track the location of the evicted pages on disk.
- Level 2 Adjustable Replacement Cache, or L2ARC- A cache residing outside of physical memory, typically on a fast SSD. It is a literal, physical extension of the RAM ARC.

The ARC Algorithm

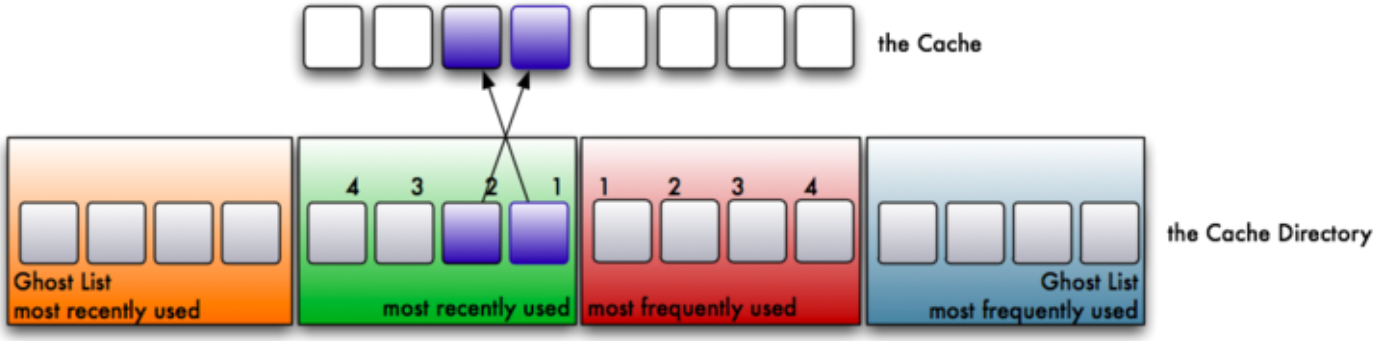
This is a simplified version of how the IBM ARC works, but it should help you understand how priority is placed both on the MRU and the MFU. First, let's assume that you have eight pages in your cache. Four pages in your cache will be used for the MRU and four pages for the MFU. Further, there will also be four pointers for the ghost MRU and four pointers for the ghost MFU. As such, the cache directory will reference 16 pages of live or evicted cache.



1. As would be expected, when block A is read from the filesystem, it will be cached in the MRU. An index pointer in the cache directory will reference the the MRU page.

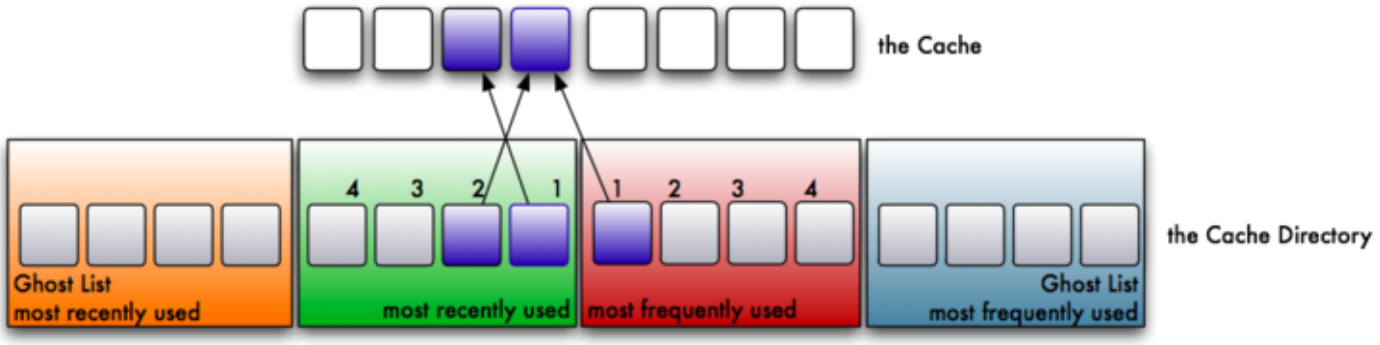


2. Suppose now a different block (block B) is read from the filesystem. It too will be cached in the MRU, and an index pointer in the cache directory will reference the second MRU page. Because block B was read more recently than block A, it gets higher preference in the MRU cache than block A. There are now two pages in the MRU cache.

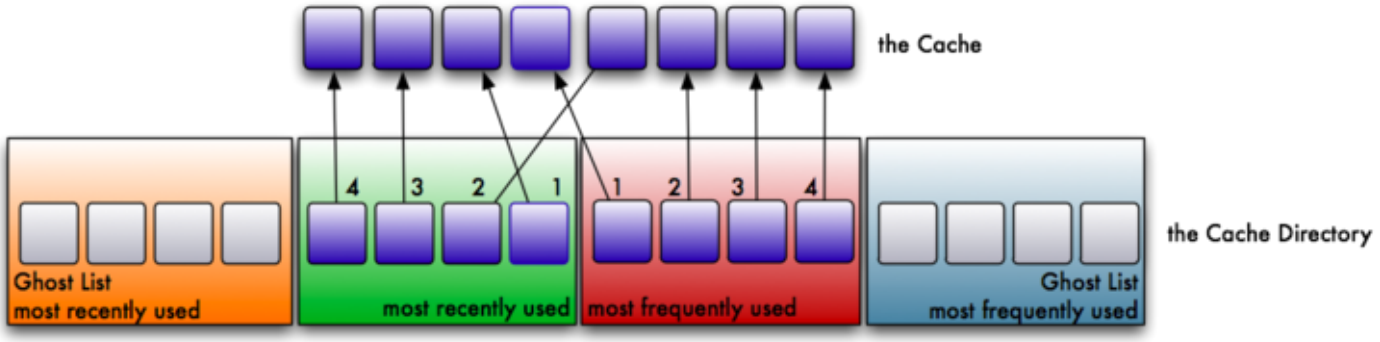


3. Now suppose block A is read again from the filesystem. This would be two reads for block A. As a result, it has been read frequently, so it will be store in the MFU. A block must be read at least twice to be stored here. Further, it is also a recent request. So, not only is the block cached in the MFU, it is also referenced in the MRU

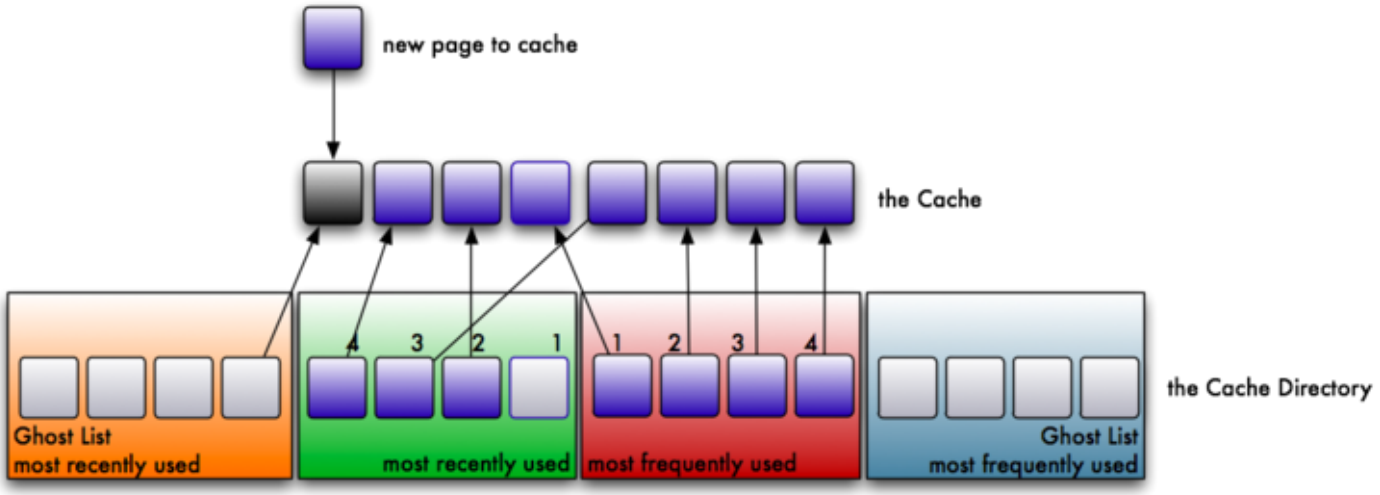
of the cache directory. As a result, although two pages reside in cache, there are three pointers in the cache directory pointing to two blocks in the cache.



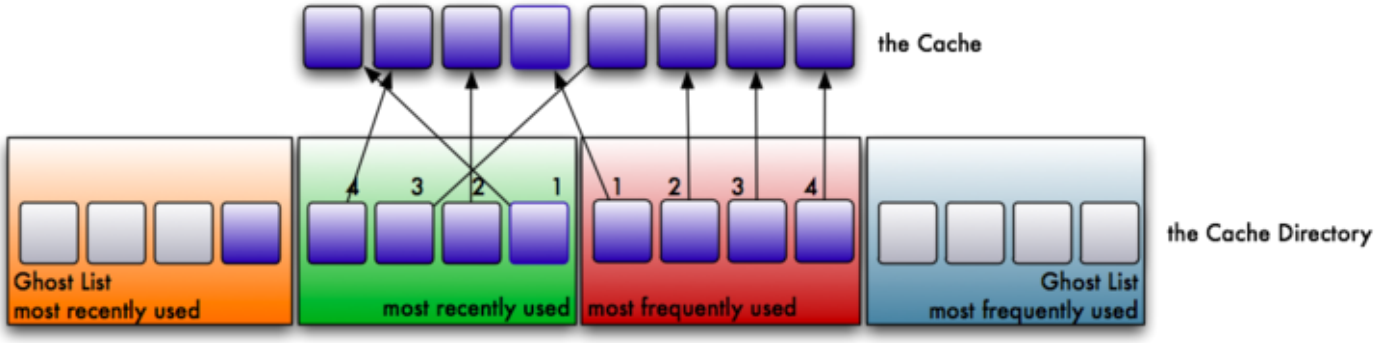
4. Eventually, the cache is filled with the above steps, and we have pointers in the MRU and the MFU of the cache directory.



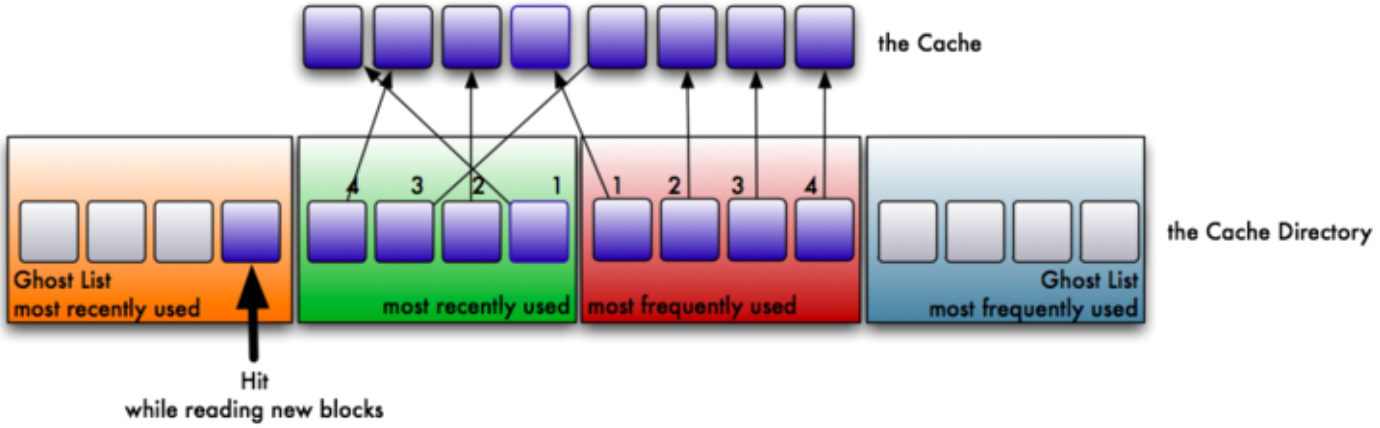
5. Here's where things get interesting. Suppose we now need to read a new block from the filesystem that is not cached. Because of the pigeon hole principle, we have more pages to cache than we can store. As such, we will need to evict a page from the cache. The oldest page in the MRU (referred to as the Least Recently Used- LRU) gets the eviction notice, and is referenced by the ghost MRU. A new page will now be available in the MRU for the newly read block.



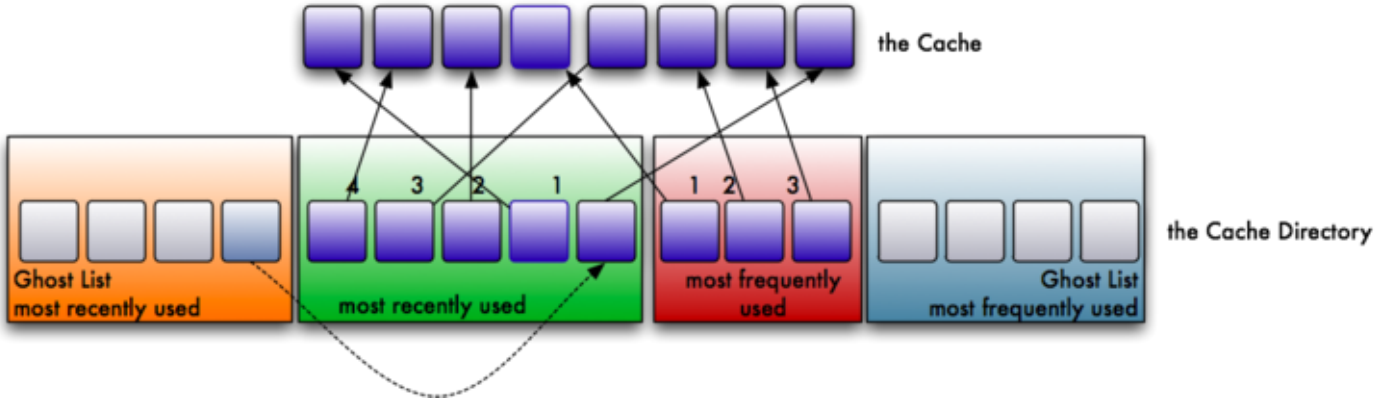
6. After the newly read block is read from the filesystem, as expected, it is stored in the MRU and referenced accordingly. Thus, we have a ghost MRU page reference, and a filled cache.



7. Just to throw a monkey wrench into the whole process, let us suppose that the recently evicted page is re-read from the filesystem. Because the ghost MRU knows it was recently evicted from the cache, we refer to this as "a phantom cache hit". Because ZFS knows it was recently cached, we need to bring it back into the MRU cache; not the MFU cache, because it was not referenced by the MFU ghost.



8. Unfortunately, our cache is too small to store the page. So, we must grow the MRU by one page to store the new phantom hit. However, our cache is only so large, so we must adjust the size of the MFU by one to make space for the MRU. Of course, the algorithm works in a similar manner on the MFU and ghost MFU. Phantom hits for the ghost MFU will enlarge the MFU, and shrink the MRU to make room for the new page.



So, imagine two polar opposite work loads. The first work load reads lot of random data from disk, with very little duplication. The MRU will likely make up most of the cache, while the MFU will make up very little. The cache has adjusted itself for the load the system is under. Consider the second work load, however, that continuously reads the same data over and over, with very little newly read data. In this scenario, the MFU will likely make up most of the cache, while the MRU will not. As a result, the cache has been adjusted to represent the load the system is under.

Remember our traditional caching scheme mentioned earlier? The Linux kernel uses the LRU scheme to swap cached pages to disk. As such, it always favors recent cache hits over frequent cache hits. This can have drastic consequences if you need to read a large number of blocks only once. You will swap out frequently cached pages to disk, even if your newly read data is only needed once. Thus, you could end up with THE SWAP OF DEATH, as the frequently requested pages have to come back off of disk from the swap area. With the ARC, we modify the cache to adapt to the load (thus, the reason it's called the "Adaptive Read Cache").

ZFS ARC Extensions

The previous algorithm is a simplified version of the ARC algorithm as designed by IBM. There are some extensions that ZFS has made, which I'll mention here:

- The ZFS ARC will occupy 1/2 of available RAM. However, this isn't static. If you have 32 GB of RAM in your server, this doesn't mean the cache will always be 16 GB. Rather, the total cache will adjust its size based on kernel decisions. If the kernel needs more RAM for a scheduled process, the ZFS ARC will be adjusted to make room for whatever the kernel needs. However, if there is space that the ZFS ARC can occupy, it will take it up.
- The ZFS ARC can work with multiple block sizes, whereas the IBM implementation uses static block sizes.
- Pages can be locked in the MRU or MFU to prevent eviction. The IBM implementation does not have this feature. As a result, the ZFS ARC algorithm is a bit more complex in choosing when a page actually gets evicted from the cache.

The L2ARC

The level 2 ARC, or L2ARC should be fast disk. As mentioned in my previous post about the ZIL, this should be DRAM DIMMs (not necessarily battery-backed), a fast SSD, or 10k+ enterprise SAS or FC disk. If you decide to use the same device for both your ZIL and your L2ARC, which is certainly acceptable, you should partition it such that the ZIL takes up very little space, like 512 MB or 1 GB, and give the rest to the pool as a striped (RAID-0) L2ARC. Persistence in the L2ARC is not needed, as the cache will be wiped on boot.

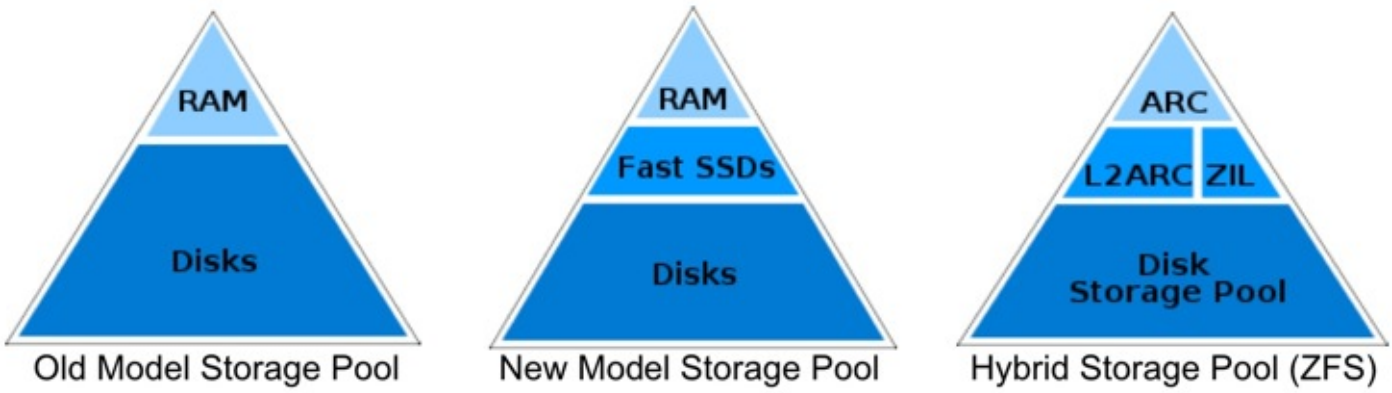


Image courtesy of [The Storage Architect]

The L2ARC is an extension of the ARC in RAM, and the previous algorithm remains untouched when an L2ARC is present. This means that as the MRU or MFU grow, they don't both simultaneously share the ARC in RAM and the L2ARC on your SSD. This would have drastic performance impacts. Instead, when a page is about to be evicted, a walking algorithm will evict the MRU and MFU pages into an 8 MB buffer, which is later set as an atomic write transaction to the L2ARC. The obvious advantage here, is that the latency of evicting pages from the cache is not impacted. Further, if a large read of data blocks is sent to the cache, the blocks are evicted before the L2ARC walk, rather than sent to the L2ARC. This minimizes polluting the L2ARC with massive sequential reads. Filling the L2ARC can also be very slow, or very fast, depending on the access to your data.

Adding an L2ARC

WARNING: Some motherboards will not present disks in a consistent manner to the Linux kernel across reboots. As such, a disk identified as /dev/sda on one boot might be /dev/sdb on the next. For the main pool where your data is stored, this is not a problem as ZFS can reconstruct the VDEVs based on the metadata geometry. For your L2ARC and SLOG devices, however, no such metadata exists. So, rather than adding them to the pool by their /dev/sd? names, you should use the /dev/disk/by-id/* names, as these are symbolic pointers to the ever-changing /dev/sd? files. If you don't heed this warning, your L2ARC device may not be added to your hybrid pool at all, and you will need to re-add it later. This could drastically affect the performance of the applications when pulling evicted pages off of disk.

You can add an L2ARC by using the "cache" VDEV. It is recommended that you stripe the L2ARC to maximize both size and speed. Persistent data is not necessary for an L2ARC, so it can be volatile DIMMs. Suppose I have 4 platter disks in my pool, and an OCZ Revodrive SSD that presents two 60 GB drives to the system. I'll partition the drives on the SSD, giving 4 GB to the ZIL and the rest to the L2ARC. This is how you would add the L2ARC to the pool. Here, I am using GNU parted to create the partitions first, then adding the SSDs. The devices in /dev/disk/by-id/ are pointing to /dev/sda and /dev/sdb. FYI.

```
# parted /dev/sda unit s mklabel gpt mkpart primary zfs 2048 4G mkpart primary zfs 4G 109418255
# parted /dev/sdb unit s mklabel gpt mkpart primary zfs 2048 4G mkpart primary zfs 4G 109418255
# zpool add tank cache \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part2 \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-X5RG0EIY7MN7676K-part2 \
log mirror \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1 \
/dev/disk/by-id/ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1
# zpool status tank
  pool: tank
 state: ONLINE
 scan: scrub repaired 0 in 1h8m with 0 errors on Sun Dec  2 01:08:26 2012
config:

        NAME                                            STATE     READ WRITE CKSUM
        tank                                            ONLINE       0     0     0
          raidz1-0                                      ONLINE       0     0     0
            sdd                                         ONLINE       0     0     0
            sde                                         ONLINE       0     0     0
            sdf                                         ONLINE       0     0     0
            sdg                                         ONLINE       0     0     0
        logs
          mirror-1                                      ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1  ONLINE       0     0     0
            ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1  ONLINE       0     0     0
        cache
          ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part2    ONLINE       0     0     0
          ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part2    ONLINE       0     0     0

errors: No known data errors
```

Further, I can check the size of the L2ARC at any time:

```
#zpool iostat -v
                                                  capacity     operations    bandwidth
pool                                            alloc   free   read  write   read  write
----------------------------------------------  -----  -----  -----  -----  -----  -----
pool                                             824G  2.82T     11     60   862K  1.05M
  raidz1                                         824G  2.82T     11     52   862K   972K
    sdd                                             -      -      5     29   289K   329K
    sde                                             -      -      5     28   289K   327K
    sdf                                             -      -      5     29   289K   329K
    sdg                                             -      -      7     35   289K   326K
logs                                                -      -      -      -      -      -
  mirror                                        1.38M  3.72G      0     19      0   277K
    ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part1    -      -      0     19      0   277K
    ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part1    -      -      0     19      0   277K
cache                                               -      -      -      -      -      -
  ata-OCZ-REVODRIVE_OCZ-69ZO5475MT43KNTU-part2  2.34G  49.8G      0      0    739  4.32K
  ata-OCZ-REVODRIVE_OCZ-9724MG8BII8G3255-part2  2.23G  49.9G      0      0    801  4.11K
----------------------------------------------  -----  -----  -----  -----  -----  -----
```

In this case, I'm using about 5 GB of cached data in the L2ARC (remember, it's striped), with plenty of space to go. In fact, the above paste is from a live running system with 32 GB of DDR2 RAM which is currently hosting this blog. The L2ARC was modified and re-added one week ago. This shows you the pace at which I am filling up the L2ARC on my system. Your situation may be different, but this isn't surprising that it is taking this long. Ben Rockwood has a Perl script that can break down the ARC and L2ARC by MRU, MFU, and the ghosts, as well as other stats. Check it out at http://www.cuddletech.com/blog/pivot/entry.php?id=979 (I don't have any experience with that script).

Conclusion

The ZFS Adjustable Replacement Cache improves on the original Adaptive Read Cache by IBM, while remaining true to the IBM design. However, the ZFS ARC has massive gains over traditional LRU and LFU caches, as deployed by the Linux kernel and other operating systems. And, with the addition of an L2ARC on fast SSD or disk, we can retrieve large amounts of data quickly, while still allowing the host kernel to adjust the memory requirements as needed.

Posted by Aaron Toponce on Friday, December 7, 2012, at 6:00 am.

Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 9 } Comments

1. John Naggets | January 23, 2015 at 11:47 am | Permalink

   When you run:

   parted /dev/sda unit s mklabel gpt mkpart primary zfs 2048 4G mkpart primary zfs 4G 109418255

   why do you start your first partition at 2048 and not at 0? any advantages? and can I do the same?

2. Aaron Toponce | January 23, 2015 at 1:19 pm | Permalink

   Starting at sector "2048" with 512 bytes per sector means the starting partition starts at the 1MB boundary. This ensures two things:

   1. That the partition is aligned on 4KB boundaries for advanced format drives.
   2. It is a software standard that partitions align on 1MB boundaries

   The previous alignment of partitions was based on the false assumption that all drives had the same cylinder alignment. However, this is horribly false. Regardless, old utilities used track alignment, rather than byte alignment, to create partitions. Of course, this really isn't an issue for your system, unless you've purchased drives that to cylinder alignment differently, at which point, performance will go in the toilet.

   You can read more about it here: http://homepage.ntlworld.com./jonathan.deboynepollard/FGA/disc-partition-alignment.html

   And yes, you can, and should use this partition alignment, if you plan on partitioning the disks.

3. John Naggets | January 26, 2015 at 4:36 am | Permalink

   Dear Aaron, first of all thank you for your complete answer!

   Now I was wondering, is this required for both "old" hard disks with 512 bytes per sector as well as newer more modern hard disks with 4 kbytes per sector?

   and why 2048 and not 4096 as start for the first partition?

4. Aaron Toponce | January 26, 2015 at 6:31 am | Permalink

Nothing is "required" for partitioning. There is nothing about old or modern drives that requires you to put your partition at sector 2048. By doing so, we improve the performance of advanced format drives, but that doesn't mean that it degrades the performance of non-AF drives. It's only moving the partition to the 1MB boundary. If you want to start your partitions at the 2MB boundary, go for it.

But, as mentioned, the reason for the change was for a deterministic and predictable way to align partition boundaries, ensure the performance of AF drives, and let all partitioning software locate use the same starting point, regardless of the geometry of the disk.

5. Vivek | January 29, 2015 at 3:50 am | Permalink

   On the 5th Diagram, under "Arc Algorithm", I am finding it a bit difficult to understand the scenario as depicted in the diagram. On point 3, you explained that Block A is read again and is therefore now a Frequently accessed. This block has two links, one in the MRU and another in the MFU. This is clear. However, on diagram 5, there are certain pages / blocks that have a link only to MFU. In what scenario will a block or page be linked only to MFU and not to MRU ? If a block is read once, it has to be on MRU and the second access onwards, it will be linked to MFU. Also, since it is recently touched, it will maintain a link to MRU as well.

   Would appreciate if you can clarify this doubt of mine. I am new to this technology and therefore, your explanation would be of great help.

6. Aaron Toponce | January 29, 2015 at 5:16 am | Permalink

   The MFU contains pages that are frequently accessed, but not necessarily recent. Using this diagram, if there are only 4 pages for the MRU and 4 pages for the MFU (for simplicity in this explanation), then the MFU could contain pages that have each been access 100 times. However, the MRU could contain pages that have only been accessed once, but more recent than the 4 pages in the MFU.

   In other words, just because a page is in the MFU, doesn't necessarily mean its recent.

   On the other hand, whenever a MFU page is accessed again, it will most certainly be in the MRU cache, as the access is recent. This is also shown in the diagram.

   So, to simplify things:

   * If a block is read, it is always added to the MRU.
   * If the same block is read again, it is a candidate for the MFU.
   * Any page read from the MFU will be added to the MRU.
   * A page read in the MRU may or may not be added to the MFU.

   Hope that answers your question.

7. Cousteau | August 8, 2015 at 4:26 pm | Permalink

   Hi,

   can you tell me please, is there a need for using ashift when adding a cache or a log device?
   When I created my pool, i set ashift=12. So what's about the SSDs I do use as cache and log devices? Are their alignments correct?

8. Anonymous | October 28, 2015 at 1:35 pm | Permalink

   In the original arc paper, if a block is hit in B1 or B2 are moved to T2, not to T1 and T2 respectively. Am I wrong?

9. asmo | August 7, 2017 at 5:09 pm | Permalink

Supposed that there are two zpools on one machine, will there be two separate ARC in memory or will the ARC cache data from both pools?

{ 8 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part V- Exporting and Importing zpools | December 10, 2012 at 6:00 am | Permalink

   [...] Our previous post finished our discussion about VDEVs by going into great detail about the ZFS ARC. Here, we'll continue our discussion about ZFS storage pools, how to migrate them across systems by exporting and importing the pools, recovering from destroyed pools, and how to upgrade your storage pool to the latest version. [...]

2. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 13, 2012 at 6:06 am | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

3. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 18, 2012 at 12:34 pm | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

4. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 18, 2012 at 12:34 pm | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

5. Aaron Toponce : Install ZFS on Debian GNU/Linux | December 20, 2012 at 8:09 am | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

6. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | January 14, 2013 at 9:16 am | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

7. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | April 19, 2013 at 4:58 am | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

8. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 23, 2013 at 7:45 am | Permalink

   [...] The Adjustable Replacement Cache (ARC) [...]

Aaron Toponce

{ 2012.12.10 }

ZFS Administration, Part V- Exporting and Importing zpools

Table of Contents

Our previous post finished our discussion about VDEVs by going into great detail about the ZFS ARC. Here, we'll continue our discussion about ZFS storage pools, how to migrate them across systems by exporting and importing the pools, recovering from destroyed pools, and how to upgrade your storage pool to the latest version.

Motivation

As a GNU/Linux storage administrator, you may come across the need to move your storage from one server to another. This could be accomplished by physically moving the disks from one storage box to another, or by copying the data from the old live running system to the new. I will cover both cases in this series. The latter deals with sending and receiving ZFS snapshots, a topic that will take us some time getting to. This post will deal with the former; that is, physically moving the drives.

One slick feature of ZFS is the ability to export your storage pool, so you can disassemble the drives, unplug their cables, and move the drives to another system. Once on the new system, ZFS gives you the ability to import the storage pool, regardless of the order of the drives. A good demonstration of this is to grab some USB sticks, plug them in, and create a ZFS storage pool. Then export the pool, unplug the sticks, drop them into a hat, and mix them up. Then, plug them back in at any random order, and re-import the pool on a new box. In fact, ZFS is smart enough to detect endianness. In other words, you can export the storage pool from a big endian system, and import the pool on a little endian system, without hiccup.

Exporting Storage Pools

When the migration is ready to take place, before unplugging the power, you need to export the storage pool. This will cause the kernel to flush all pending data to disk, writes data to the disk acknowledging that the export was done, and removes all knowledge that the storage pool existed in the system. At this point, it's safe to shut down the computer, and remove the drives.

If you do not export the storage pool before removing the drives, you will not be able to import the drives on the new system, and you might not have gotten all unwritten data flushed to disk. Even though the data will remain consistent due to the nature of the filesystem, when importing, it will appear to the old system as a faulted pool. Further, the destination system will refuse to import a pool that has not been explicitly exported. This is to prevent race conditions with network attached storage that may be already using the pool.

To export a storage pool, use the following command:

```
# zpool export tank
```

This command will attempt to unmount all ZFS datasets as well as the pool. By default, when creating ZFS storage pools and filesystems, they are automatically mounted to the system. There is no need to explicitly unmount the filesystems as you with with ext3 or ext4. The export will handle that. Further, some pools may refuse to be exported, for whatever reason. You can pass the "-f" switch if needed to force the export

Importing Storage Pools

Once the drives have been physically installed into the new server, you can import the pool. Further, the new system may have multiple pools installed, to which you will want to determine which pool to import, or to import them all. If the storage pool "tank" does not already exist on the new server, and this is the pool you wish to import, then you can run the following command:

```
# zpool import tank
# zpool status tank
 state: ONLINE
 scan: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sde     ONLINE       0     0     0
            sdf     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            sdg     ONLINE       0     0     0
            sdh     ONLINE       0     0     0
          mirror-2  ONLINE       0     0     0
            sdi     ONLINE       0     0     0
            sdj     ONLINE       0     0     0

errors: No known data errors
```

Your storage pool state may not be "ONLINE", meaning that everything is healthy. If the system does not recognize a disk in your pool, you may get a "DEGRADED" state. If one or more of the drives appear as faulty to the system, then you may get a "FAULTED" state in your pool. You will need to troubleshoot what drives are causing the problem, and fix accordingly.

You can import multiple pools simultaneously by either specifying each pool as an argument, or by passing the "-a" switch for importing all discovered pools. For importing the two pools "tank1" and "tank2", type:

```
# zpool import tank1 tank2
```

For importing all known pools, type:

```
# zpool import -a
```

Recovering A Destroyed Pool

If a ZFS storage pool was previously destroyed, the pool can still be imported to the system. Destroying a pool doesn't wipe the data on the disks, so the metadata is still in tact, and the pool can still be discovered. Let's take a clean pool called "tank", destroy it, move the disks to a new system, then try to import the pool. You will need to pass the "-D" switch to tell ZFS to import a destroyed pool. Do not provide the pool name as an argument, as you would normally do:

```
(server A)# zpool destroy tank
(server B)# zpool import -D
   pool: tank
     id: 17105118590326096187
  state: ONLINE (DESTROYED)
 action: The pool can be imported using its name or numeric identifier.
 config:

        tank        ONLINE
          mirror-0  ONLINE
            sde     ONLINE
            sdf     ONLINE
          mirror-1  ONLINE
            sdg     ONLINE
            sdh     ONLINE
          mirror-2  ONLINE
            sdi     ONLINE
            sdj     ONLINE

   pool: tank
     id: 2911384395464928396
  state: UNAVAIL (DESTROYED)
 status: One or more devices are missing from the system.
 action: The pool cannot be imported. Attach the missing
        devices and try again.
   see: http://zfsonlinux.org/msg/ZFS-8000-6X
 config:

        tank          UNAVAIL  missing device
          sdk         ONLINE
          sdr         ONLINE

        Additional devices are known to be part of this pool, though their
        exact configuration cannot be determined.
```

Notice that the state of the pool is "ONLINE (DESTROYED)". Even though the pool is "ONLINE", it is only partially online. Basically, it's only been discovered, but it's not available for use. If you run the "df" command, you will find that the storage pool is not mounted. This means the ZFS filesystem datasets are not available, and you currently cannot store data into the pool. However, ZFS has found the pool, and you can bring it fully ONLINE for standard usage by running the import command one more time, this time specifying the pool name as an argument to import:

```
(server B)# zpool import -D tank
cannot import 'tank': more than one matching pool
import by numeric ID instead
(server B)# zpool import -D 17105118590326096187
(server B)# zpool status tank
  pool: tank
 state: ONLINE
 scan: none requested
config:
```

```
        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sde     ONLINE       0     0     0
            sdf     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            sdg     ONLINE       0     0     0
            sdh     ONLINE       0     0     0
          mirror-2  ONLINE       0     0     0
            sdi     ONLINE       0     0     0
            sdj     ONLINE       0     0     0

errors: No known data errors
```

Notice that ZFS was warning me that it found more than on storage pool matching the name "tank", and to import the pool, I must use its unique identifier. So, I pass that as an argument from my previous import. This is because in my previous output, we can see there are two known pools with the pool name "tank". However, after specifying its ID, I was able to successfully bring the storage pool to full "ONLINE" status. You can identify this by checking its status:

```
# zpool status tank
  pool: tank
 state: ONLINE
status: The pool is formatted using an older on-disk format.  The pool can
        still be used, but some features are unavailable.
action: Upgrade the pool using 'zpool upgrade'.  Once this is done, the
        pool will no longer be accessible on older software versions.
 scrub: none requested
config:

        NAME        STATE     READ WRITE CKSUM
        tank        ONLINE       0     0     0
          mirror-0  ONLINE       0     0     0
            sde     ONLINE       0     0     0
            sdf     ONLINE       0     0     0
          mirror-1  ONLINE       0     0     0
            sdg     ONLINE       0     0     0
            sdh     ONLINE       0     0     0
          mirror-2  ONLINE       0     0     0
            sdi     ONLINE       0     0     0
            sdj     ONLINE       0     0     0
```

Upgrading Storage Pools

One thing that may crop up when migrating disk, is that there may be different pool and filesystem versions of the software. For example, you may have exported the pool on a system running pool version 20, while importing into a system with pool version 28 support. As such, you can upgrade your pool version to use the latest software for that release. As is evident with the previous example, it seems that the new server has an update version of the software. I am going to upgrade.

WARNING: Once you upgrade your pool to a newer version of ZFS, older versions will not be able to use the storage pool. So, make sure that when you upgrade the pool, you know that there will be no need for going back to the old system. Further, there is no way to revert the upgrade and revert to the old version.

First, we can see a brief description of features that will be available to the pool:

```
# zpool upgrade -v
This system is currently running ZFS pool version 28.
```

```
The following versions are supported:

VER  DESCRIPTION
---  --------------------------------------------------------
 1   Initial ZFS version
 2   Ditto blocks (replicated metadata)
 3   Hot spares and double parity RAID-Z
 4   zpool history
 5   Compression using the gzip algorithm
 6   bootfs pool property
 7   Separate intent log devices
 8   Delegated administration
 9   refquota and refreservation properties
 10  Cache devices
 11  Improved scrub performance
 12  Snapshot properties
 13  snapused property
 14  passthrough-x aclinherit
 15  user/group space accounting
 16  stmf property support
 17  Triple-parity RAID-Z
 18  Snapshot user holds
 19  Log device removal
 20  Compression using zle (zero-length encoding)
 21  Deduplication
 22  Received properties
 23  Slim ZIL
 24  System attributes
 25  Improved scrub stats
 26  Improved snapshot deletion performance
 27  Improved snapshot creation performance
 28  Multiple vdev replacements

For more information on a particular version, including supported releases,
see the ZFS Administration Guide.
```

So, let's perform the upgrade to get to version 28 of the pool:

```
# zpool upgrade -a
```

As a sidenote, when using ZFS on Linux, the RPM and Debian packages will contain an /etc/init.d/zfs init script for setting up the pools and datasets on boot. This is done by importing them on boot. However, at shutdown, the init script does not export the pools. Rather, it just unmounts them. So, if you migrate the disk to another box after only shutting down, you will be not be able to import the storage pool on the new box.

Conclusion

There are plenty of situations where you may need to move disk from one storage server to another. Thankfully, ZFS makes this easy with exporting and importing pools. Further, the "zpool" command has enough subcommands and switches to handle the most common scenarios when a pool will not export or import. Towards the very end of the series, I'll discuss the "zdb" command, and how it may be useful here. But at this point, steer clear of zdb, and just focus on keeping your pools in order, and properly exporting and importing them as needed.

Posted by Aaron Toponce on Monday, December 10, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment

or trackback from your blog. For IM, Email or Microblogs, here is

the Shortlink.

{ 4 } Comments

1. eagle275 | February 18, 2013 at 7:58 am | Permalink

   I run my storage server on OpenIndiana at the moment. But I want to switch over to (K)unbutu and keep using my zpool using your nice instructions
   Should I "export" my pool vom OpenIndiana first, then setup Linux , reinstall ZFS support and import the pool or is exporting not needed in that case?

2. Joe Finley | October 23, 2013 at 9:23 am | Permalink

   Let's say I wanted to ZFS send to another location, but practical bandwidth to ZFS SEND would take weeks/months, is it possible to seed onto a USB drive and import at another location?

3. Aaron Toponce | October 23, 2013 at 10:26 am | Permalink

   Sure. Just make a 1-drive ZFS pool, send to it, then when you get to the other location, send from that drive to the final destination.

4. fmyhr | July 31, 2016 at 4:13 pm | Permalink

   Thank you for your ZFS guide!

   Could you please correct your statement on this page that
   # zpool import tank1 tank2
   imports both pools tank1 and tank2? According to man zpool and my own experience on ZoL, this command instead *RENAMES* tank1 to tank2. That was actually what I wanted, and it worked, but can imagine it'd be a nasty surprise for others following your guide.

{ 7 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part IV- The Adjustable Replacement Cache | December 18, 2012 at 12:34 pm | Permalink

   [...] Exporting and Importing Storage Pools [...]

2. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 20, 2012 at 8:07 am | Permalink

   [...] Exporting and Importing Storage Pools [...]

3. Aaron Toponce : ZFS Administration, Part VI- Scrub and Resilver | December 20, 2012 at 8:09 am | Permalink

   [...] Exporting and Importing Storage Pools [...]

4. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 29, 2012 at 6:19 am | Permalink

   [...] Exporting and Importing Storage Pools [...]

5. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 29, 2012 at 6:23 am | Permalink

[...] Exporting and Importing Storage Pools [...]

6. Aaron Toponce : Install ZFS on Debian GNU/Linux | January 7, 2013 at 9:27 pm | Permalink

   [...] Exporting and Importing Storage Pools [...]

7. Aaron Toponce : ZFS Administration, Part XIV- ZVOLS | April 19, 2013 at 4:59 am | Permalink

   [...] Exporting and Importing Storage Pools [...]

Aaron Toponce

{ 2012.12.11 }

ZFS Administration, Part VI- Scrub and Resilver

Table of Contents

Standard Validation

In GNU/Linux, we have a number of filesystem checking utilities for verifying data integrity on the disk. This is done through the "fsck" utility. However, it has a couple major drawbacks. First, you must fsck the disk offline if you are intending on fixing data errors. This means downtime. So, you must use the "umount" command to unmount your disks, before the fsck. For root partitions, this further means booting from another medium, like a CDROM or USB stick. Depending on the size of the disks, this downtime could take hours. Second, the filesystem, such as ext3 or ext4, knows nothing of the underlying data structures, such as LVM or RAID. You may only have a bad block on one disk, but a good block on another disk. Unfortunately, Linux software RAID has no idea which is good or bad, and from the perspective of ext3 or ext4, it will get good data if read from the disk containing the good block, and corrupted data from the disk containing the bad block, without any control over which disk to pull the data from, and fixing the corruption. These errors are known as "silent data errors", and there is really nothing you can do about it with the standard GNU/Linux filesystem stack.

ZFS Scrubbing

With ZFS on Linux, detecting and correcting silent data errors is done through scrubbing the disks. This is similar in technique to ECC RAM, where if an error resides in the ECC DIMM, you can find another register that contains the good data, and use it to fix the bad register. This is an old technique that has been around for a while, so it's surprising that it's not available in the standard suite of journaled filesystems. Further, just like you can scrub ECC RAM on a live running system, without downtime, you should be able to scrub your disks without downtime as well. With ZFS, you can.

While ZFS is performing a scrub on your pool, it is checking every block in the storage pool against its known checksum. Every block from top-to-bottom is checksummed using an appropriate algorithm by default. Currently, this is the "fletcher4" algorithm, which is a 256-bit algorithm, and it's *fast*. This can be changed to using the SHA-256 algorithm, although it may not recommended, as calculating the SHA-256 checksum is more costly than fletcher4. However, because of SHA-256, you have a 1 in 2^256 or 1 in 10^77 chance that a corrupted block hashes to the same SHA-256 checksum. This is a 0.00000000000000000000000000000000000000000000000000000000000000000000000000000001% chance. For reference, uncorrected ECC memory errors will happen on about 50 orders of magnitude more frequently, with the most reliable hardware on the market. So, when scrubbing your data, the probability is that either the checksum will match, and you have a good data block, or it won't match, and you have a corrupted data block.

Scrubbing ZFS storage pools is not something that happens automatically. You need to do it manually, and it's highly recommended that you do it on a regularly scheduled interval. The recommended frequency at which you should scrub the data depends on the quality of the underlying disks. If you have SAS or FC disks, then once per month should be sufficient. If you have consumer grade SATA or SCSI, you should do once per week. You can schedule a scrub easily with the following command:

```
# zpool scrub tank
# zpool status tank
```

```
  pool: tank
 state: ONLINE
 scan: scrub in progress since Sat Dec  8 08:06:36 2012
    32.0M scanned out of 48.5M at 16.0M/s, 0h0m to go
    0 repaired, 65.99% done
config:

        NAME         STATE     READ WRITE CKSUM
        tank         ONLINE       0     0     0
          mirror-0   ONLINE       0     0     0
            sde      ONLINE       0     0     0
            sdf      ONLINE       0     0     0
          mirror-1   ONLINE       0     0     0
            sdg      ONLINE       0     0     0
            sdh      ONLINE       0     0     0
          mirror-2   ONLINE       0     0     0
            sdi      ONLINE       0     0     0
            sdj      ONLINE       0     0     0

errors: No known data errors
```

As you can see, you can get a status of the scrub while it is in progress. Doing a scrub can severely impact performance of the disks and the applications needing them. So, if for any reason you need to stop the scrub, you can pass the "-s" switch to the scrub subcommand. However, you should let the scrub continue to completion.

```
# zpool scrub -s tank
```

You should put something similar to the following in your root's crontab, which will execute a scrub every Sunday at 02:00 in the morning:

```
0 2 * * 0 /sbin/zpool scrub tank
```

Self Healing Data

If your storage pool is using some sort of redundancy, then ZFS will not only detect the silent data errors on a scrub, but it will also correct them if good data exists on a different disk. This is known as "self healing", and can be demonstrated in the following image. In my RAIDZ post, I discussed how the data is self-healed with RAIDZ, using the parity and a reconstruction algorithm. I'm going to simplify it a bit, and use just a two way mirror. Suppose that an application needs some data blocks, and in those blocks, on of them is corrupted. How does ZFS know the data is corrupted? By checking the SHA-256 checksum of the block, as already mentioned. If a checksum does not match on a block, it will look at my other disk in the mirror to see if a good block can be found. If so, the good block is passed to the application, then ZFS will fix the bad block in the mirror, so that it also passes the SHA-256 checksum. As a result, the application will always get good data, and your pool will always be in a good, clean, consistent state.

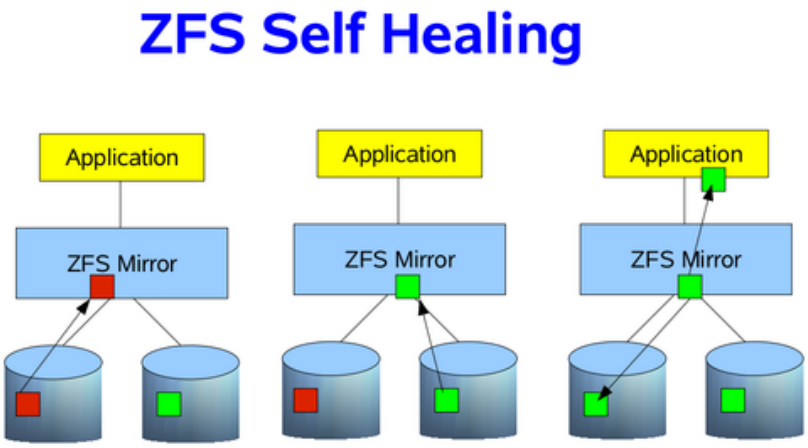


Image courtesy of root.cz, showing how ZFS self heals data.

Resilvering Data

Resilvering data is the same concept as rebuilding or resyncing data onto the new disk into the array. However, with Linux software RAID, hardware RAID controllers, and other RAID implementations, there is no distinction between which blocks are actually live, and which aren't. So, the rebuild starts at the beginning of the disk, and does not stop until it reaches the end of the disk. Because ZFS knows about the the RAID structure and the filesystem metadata, we can be smart about rebuilding the data. Rather than wasting our time on free disk, where live blocks are not stored, we can concern ourselves with ONLY those live blocks. This can provide

significant time savings, if your storage pool is only partially filled. If the pool is only 10% filled, then that means only working on 10% of the drives. Win. Thus, with ZFS we need a new term than "rebuilding", "resyncing" or "reconstructing". In this case, we refer to the process of rebuilding data as "resilvering".

Unfortunately, disks will die, and need to be replaced. Provided you have redundancy in your storage pool, and can afford some failures, you can still send data to and receive data from applications, even though the pool will be in "DEGRADED" mode. If you have the luxury of hot swapping disks while the system is live, you can replace the disk without downtime (lucky you). If not, you will still need to identify the dead disk, and replace it. This can be a chore if you have many disks in your pool, say 24. However, most GNU/Linux operating system vendors, such as Debian or Ubuntu, provide a utility called "hdparm" that allows you to discover the serial number of all the disks in your pool. This is, of course, that the disk controllers are presenting that information to the Linux kernel, which they typically do. So, you could run something like:

```
# for i in a b c d e f g; do echo -n "/dev/sd$i: "; hdparm -I /dev/sd$i | awk '/Serial Number/ {print $3}'; done
/dev/sda: OCZ-9724MG8BII8G3255
/dev/sdb: OCZ-69ZO5475MT43KNTU
/dev/sdc: WD-WCAPD3307153
/dev/sdd: JP2940HD0K9RJC
/dev/sde: /dev/sde: No such file or directory
/dev/sdf: JP2940HD0SB8RC
/dev/sdg: S1D1C3WR
```

It appears that /dev/sde is my dead disk. I have the serial numbers for all the other disks in the system, but not this one. So, by process of elimination, I can go to the storage array, and find which serial number was not printed. This is my dead disk. In this case, I find serial number "JP2940HD01VLMC". I pull the disk, replace it with a new one, and see if /dev/sde is repopulated, and the others are still online. If so, I've found my disk, and can add it to the pool. This has actually happened to me twice already, on both of my personal hypervisors. It was a snap to replace, and I was online in under 10 minutes.

To replace an dead disk in the pool with a new one, you use the "replace" subcommand. Suppose the new disk also identifed itself as /dev/sde, then I would issue the following command:

```
# zpool replace tank sde sde
# zpool status tank
  pool: tank
 state: ONLINE
status: One or more devices is currently being resilvered.  The pool will
        continue to function, possibly in a degraded state.
action: Wait for the resilver to complete.
 scrub: resilver in progress for 0h2m, 16.43% done, 0h13m to go
config:

        NAME          STATE       READ WRITE CKSUM
        tank          DEGRADED       0     0     0
          mirror-0    DEGRADED       0     0     0
            replacing DEGRADED       0     0     0
            sde       ONLINE         0     0     0
            sdf       ONLINE         0     0     0
          mirror-1    ONLINE         0     0     0
            sdg       ONLINE         0     0     0
            sdh       ONLINE         0     0     0
          mirror-2    ONLINE         0     0     0
            sdi       ONLINE         0     0     0
            sdj       ONLINE         0     0     0
```

The resilver is analagous to a rebuild with Linux software RAID. It is rebuilding the data blocks on the new disk until the mirror, in this case, is in a completely healthy state. Viewing the status of the resilver will help you get an idea of when it will complete.

Identifying Pool Problems

Determining quickly if everything is functioning as it should be, without the full output of the "zpool status" command can be done by passing the "-x" switch. This is useful for scripts to parse without fancy logic, which could alert you in the event of a failure:

```
# zpool status -x
all pools are healthy
```

The rows in the "zpool status" command give you vital information about the pool, most of which are self-explanatory. They are defined as follows:

- pool- The name of the pool.
- state- The current health of the pool. This information refers only to the ability of the pool to provide the necessary replication level.

- status- A description of what is wrong with the pool. This field is omitted if no problems are found.
- action- A recommended action for repairing the errors. This field is an abbreviated form directing the user to one of the following sections. This field is omitted if no problems are found.
- see- A reference to a knowledge article containing detailed repair information. Online articles are updated more often than this guide can be updated, and should always be referenced for the most up-to-date repair procedures. This field is omitted if no problems are found.
- scrub- Identifies the current status of a scrub operation, which might include the date and time that the last scrub was completed, a scrub in progress, or if no scrubbing was requested.
- errors- Identifies known data errors or the absence of known data errors.
- config- Describes the configuration layout of the devices comprising the pool, as well as their state and any errors generated from the devices. The state can be one of the following: ONLINE, FAULTED, DEGRADED, UNAVAILABLE, or OFFLINE. If the state is anything but ONLINE, the fault tolerance of the pool has been compromised.

The columns in the status output, "READ", "WRITE" and "CHKSUM" are defined as follows:

- NAME- The name of each VDEV in the pool, presented in a nested order.
- STATE- The state of each VDEV in the pool. The state can be any of the states found in "config" above.
- READ- I/O errors occurred while issuing a read request.
- WRITE- I/O errors occurred while issuing a write request.
- CHKSUM- Checksum errors. The device returned corrupted data as the result of a read request.

Conclusion

Scrubbing your data on regular intervals will ensure that the blocks in the storage pool remain consistent. Even though the scrub can put strain on applications wishing to read or write data, it can save hours of headache in the future. Further, because you could have a "damaged device" at any time (see http://docs.oracle.com/cd/E19082-01/817-2271/gbbvf/index.html about damaged devices with ZFS), properly knowing how to fix the device, and what to expect when replacing one, is critical to storage administration. Of course, there is plenty more I could discuss about this topic, but this should at least introduce you to the concepts of scrubbing and resilvering data.

Posted by Aaron Toponce on Tuesday, December 11, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 15 } Comments

1. Anca Emanuel | December 11, 2012 at 11:51 am | Permalink

   This is wrestling with the octopus ??
   Conclusion: the need for simple standard administation was ignored.

2. Aaron Toponce | December 11, 2012 at 12:45 pm | Permalink

   Huh?

3. eagle275 | February 18, 2013 at 8:36 am | Permalink

   I believe he meant :

   Put (self adhesive) Labels on each device , that contain Serial Number AND to which device the disk was "discovered" by linux - so you should have known "oh sde fails" - look at your array, find attached label "sde" and you have the missing disk, without trial and error

4. Graham Perrin | February 28, 2013 at 12:58 am | Permalink

   zpool status -x

   "… will return “all pools are healthy” even if one device is failed in a RAIDZ pool. In the other words, your data is healthy doesn’t mean all devices in your pool are healthy. So go with “zpool status” at any time. …"

   https://diigo.com/0x79w for highlights from http://icesquare.com/wordpress/how-to-improve-zfs-performance/

5. Ryan | July 25, 2013 at 2:24 pm | Permalink

   Can you elaborate on what "then ZFS will fix the bad block in the mirror" entails? (Self Healing Data section)

   Does this mean that ZFS relocates the data from the bad block to another block on that device? At the same time, does the SMART function of the drive mark that sector as bad and relocate it to a spare, so it isn't written to again?

6. Aaron Toponce | August 7, 2013 at 10:11 am | Permalink

   First off, ZFS doesn't have any built in SMART functionality. If you want SMART, you need to install the smartmontools package for your operating system. Second, when self healing the data, due to the COW nature of the filesystem, the healed block will be in a physically different location, with the metadata and uberblock updated. ZFS does have the capability of knowing where bad blocks exist on the filesystem, and not writing to them again in the future.

7. Ryan | September 15, 2013 at 8:51 am | Permalink

   Wow... what does zfs not do?

   Thanks for writing this blog on zfs, by the way. It's well written and easy to follow. It convinced me to switch!

8. ianjo | September 29, 2013 at 10:59 am | Permalink

   You state that the default checksum algorithm is sha-256, but searching on the internet I believe that no zfs implementation uses sha-256 by default. I'm using zfs 0.6.2 on linux and the manpage states:
   Controls the checksum used to verify data integrity. The default value is on, which automati−
   cally selects an appropriate algorithm (currently, fletcher4, but this may change in future
   releases). The value off disables integrity checking on user data. Disabling checksums is NOT a
   recommended practice.

   To interested readers, this can be changed with zfs set checksum=sha256 -- I do it for all my newly-created filesystems.

9. Torge Kummerow | July 10, 2014 at 1:08 pm | Permalink

   What is the behaviour, if the corruption happens in the SHA Hash?

10. Aaron Toponce | July 11, 2014 at 9:37 am | Permalink

    Each node in the merkle tree is also SHA256 checksummed, all the way up to the root node. 128 Merkle tree revisions are kept, before reusing. As such, if a SHA256 checksum is corrupted, that leaf in the node is bad, and can be fixed, provided rendundancy, by looking at the checksum in the parent leaf of the tree, and rebuilding based on the redundancy in the pool.

11. Francois Scheurer | October 6, 2014 at 3:19 am | Permalink

    "Unfortunately, Linux software RAID has no idea which is good or bad, and from the perspective of ext3 or ext4, it will get good data if read from the disk containing the good block, and corrupted data from the disk containing the bad block, without any control over which disk to pull the data from, and fixing the corruption. These errors are known as "silent data errors", and there is really nothing you can do about it with the standard GNU/Linux filesystem stack."

    I think this is not completly correct:
    Linux and ext3/4 does not store block checksums (unfortunately..), but hard disk controllers write ECC codes along the data.
    When reading errors occurs, they will be detected and if possible corrected.
    If correction with ECC is not possible, then the linux kernel will see an unrecoverable read error and with software raid (mdadm) it will then re-read the block from another raid replica (other disk).
    Then it will overwrite the bad block with the correct data, like with the zfs scrubbing, but with the advantage of doing it on demand instead of having to scrub the huge whole disks.
    If the overwrite fails (write error), then it will automatically put the disk as offline and send an email alert.

    We are currently looking for a similar behavior on zfs pools, because right now we are seeing read errors with zfs pool on freebsd but unfortunately the bad disls stay online until some sysadmin put them manually offline...

    "However, with Linux software RAID, hardware RAID controllers, and other RAID implementations, there is no distinction between which blocks are actually live, and which aren't. So, the rebuild starts at the beginning of the disk, and does not stop until it reaches the end of the disk. Because ZFS knows about the the RAID structure and the filesystem metadata, we can be smart about rebuilding the data. Rather than wasting our time on free disk, where live blocks are not stored, we can concern ourselves with ONLY those live blocks."

Yes Linux mdadm will know nothing about fs usage and free blocks/inodes.
But a cool feature of mdadm is the bitmap that allows a resync (after a power loss for example) to only resync the modified/unsynchronized blocks of the raid.

12. JeanDo | March 24, 2015 at 11:52 am | Permalink

    Better raidz3 or spares ?

    I have a brand new server with 8 disks. Same model/date/size. I have three options (among plenty of others):

    1. Make a raidz3 5+3 pool.
    2. Make a raid 0+1 pool: "mirror sda sdb mirror sdc sdd mirror sde sdf mirror sdg sdh"
    3. Make a raid 0+z1 : "raidz1 sda sdb sdc raidz1 sdd sde sdf spare sdg sdh"

    The question is: what is the safer in the long run ?

    I feel that with schemes 1 or 2, all disks will worn out at the same speed, and might tend to crash at about the same date. While with scheme 3, the two disks will be in vacation until the first replacement, and will have a fresh history then...

    Do you have anything in your experience pro/against that ? Any advice ?

13. santosh loke | July 26, 2015 at 5:17 am | Permalink

    very well written and explained.Thanks for the great efforts!

14. remya | February 29, 2016 at 6:54 pm | Permalink

    Hi, Recently we faced an issue in ZFS , there was one pool was in degraded state.

    We tried to switch over to another ZFS, but still the same issue. occured?

    what could be the cuase

    Thanks and Regards
    Remya

15. Colbyu | December 30, 2016 at 3:00 pm | Permalink

    I would actually recommend, when creating the pool, to add the diskd by /dev/disk/by-id rather than using the sd_ node because that id will then be used as the disk name in the zpool status view. The id typically contains the disc's serial number, so you will see immediately which disk is bad. It's also considered generally a good practice, less ambiguous, and can alleviate an issue that can occur when importing the array on a new system.

{ 13 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part VII- Zpool Properties | December 12, 2012 at 6:01 am | Permalink

   [...] from our last post on scrubbing and resilvering data in zpools, we move on to changing properties in the [...]

2. Aaron Toponce: ZFS Administration, Part IX- Copy-on-write - Bartle Doo | December 14, 2012 at 11:01 am | Permalink

   [...] ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and [...]

3. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | December 18, 2012 at 12:34 pm | Permalink

   [...] Scrub and Resilver [...]

4. Aaron Toponce: ZFS Administration, Part XI- Compression and Deduplication - Bartle Doo | December 18, 2012 at 4:40 pm | Permalink

   [...] ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and [...]

5. Aaron Toponce: ZFS Administration, Part XIII- Sending and Receiving Filesystems - Bartle Doo | December 20, 2012 at 2:32 pm | Permalink

[...] ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and [...]

6. Aaron Toponce: ZFS Administration, Part XIV- ZVOLS - Bartle Doo | December 21, 2012 at 11:02 am | Permalink

   [...] ZFS Intent Log (ZIL) The Adjustable Replacement Cache (ARC) Exporting and Importing Storage Pools Scrub and Resilver Getting and Setting Properties Best Practices and [...]

7. Aaron Toponce : ZFS Administration, Part XV- iSCSI, NFS and Samba | January 7, 2013 at 9:19 pm | Permalink

   [...] Scrub and Resilver [...]

8. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | January 7, 2013 at 9:24 pm | Permalink

   [...] Scrub and Resilver [...]

9. Aaron Toponce : Install ZFS on Debian GNU/Linux | April 19, 2013 at 4:54 am | Permalink

   [...] Scrub and Resilver [...]

10. Aaron Toponce : ZFS Administration, Part I- VDEVs | April 19, 2013 at 4:55 am | Permalink

    [...] Scrub and Resilver [...]

11. Aaron Toponce : ZFS Administration, Part XI- Compression and Deduplication | April 19, 2013 at 4:58 am | Permalink

    [...] Scrub and Resilver [...]

12. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | April 19, 2013 at 4:58 am | Permalink

    [...] Scrub and Resilver [...]

13. Aaron Toponce : ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL) | April 19, 2013 at 5:02 am | Permalink

    [...] Scrub and Resilver [...]

Aaron Toponce

{ 2012.12.12 }

ZFS Administration, Part VII- Zpool Properties

Table of Contents

Continuing from our last post on scrubbing and resilvering data in zpools, we move on to changing properties in the zpool.

Motivation

With ext4, and many filesystems in GNU/Linux, we have a way for tuning various flags in the filesystem. Things like setting labels, default mount options, and other tunables. With ZFS, it's no different, and in fact, is far more verbose. These properties allow us to modify all sorts of variables, both for the pool, and for the datasets it contains. Thus, we can "tune" the filesystem to our liking or needs. However, not every property is tunable. Some are read-only. But, we'll define what each of the properties are and how they affect the pool. Note, we are only looking at zpool properties, and we will get to ZFS dataset properties when we reach the dataset subtopic.

Zpool Properties

- allocated: The amount of data that has been committed into the pool by all of the ZFS datasets. This setting is read-only.
- altroot: Identifies an alternate root directory. If set, this directory is prepended to any mount points within the pool. This property can be used when examining an unknown pool, if the mount points cannot be trusted, or in an alternate boot environment, where the typical paths are not valid.Setting altroot defaults to using "cachefile=none", though this may be overridden using an explicit setting.
- ashift: **Can only be set at pool creation time.** Pool sector size exponent, to the power of 2. I/O operations will be aligned to the specified size boundaries. Default value is "9", as 2^9 = 512, the standard sector size

operating system utilities use for reading and writing data. For advanced format drives with 4 KiB boundaries, the value should be set to "ashift=12", as 2^12 = 4096.

- autoexpand: **Must be set before replacing the first drive in your pool.** Controls automatic pool expansion when the underlying LUN is grown. Default is "off". After all drives in the pool have been replaced with larger drives, the pool will automatically grow to the new size. This setting is a boolean, with values either "on" or "off".
- autoreplace: Controls automatic device replacement of a "spare" VDEV in your pool. **Default is set to "off".** As such, device replacement must be initiated manually by using the "zpool replace" command. This setting is a boolean, with values either "on" or "off".
- bootfs: Read-only setting that defines the bootable ZFS dataset in the pool. This is typically set by an installation program.
- cachefile: Controls the location of where the pool configuration is cached. When importing a zpool on a system, ZFS can detect the drive geometry using the metadata on the disks. However, in some clustering environments, the cache file may need to be stored in a different location for pools that would not automatically be imported. Can be set to any string, but for most ZFS installations, the default location of "/etc/zfs/zpool.cache" should be sufficient.
- capacity: Read-only value that identifies the percentage of pool space used.
- comment: A text string consisting of no more than 32 printable ASCII characters that will be stored such that it is available even if the pool becomes faulted. An administrator can provide additional information about a pool using this setting.
- dedupditto: Sets a block deduplication threshold, and if the reference count for a deduplicated block goes above the threshold, a duplicate copy of the block is stored automatically. The default value is 0. Can be any positive number.
- dedupratio: Read-only deduplication ratio specified for a pool, expressed as a multiplier
- delegation: Controls whether a non-privileged user can be granted access permissions that are defined for the dataset. The setting is a boolean, defaults to "on" and can be "on" or "off".
- expandsize: Amount of uninitialized space within the pool or device that can be used to increase the total capacity of the pool. Uninitialized space consists of any space on an EFI labeled vdev which has not been brought online (i.e. "zpool online -e"). This space occurs when a LUN is dynamically expanded.
- failmode: Controls the system behavior in the event of catastrophic pool failure. This condition is typically a result of a loss of connectivity to the underlying storage device(s) or a failure of all devices within the pool. The behavior of such an event is determined as follows:
    - wait: Blocks all I/O access until the device connectivity is recovered and the errors are cleared. This is the default behavior.
    - continue: Returns EIO to any new write I/O requests but allows reads to any of the remaining healthy devices. Any write requests that have yet to be committed to disk would be blocked.
    - panic: Prints out a message to the console and generates a system crash dump.
- free: Read-only value that identifies the number of blocks within the pool that are not allocated.
- guid: Read-only property that identifies the unique identifier for the pool. Similar to the UUID string for ext4 filesystems.
- health: Read-only property that identifies the current health of the pool, as either ONLINE, DEGRADED, FAULTED, OFFLINE, REMOVED, or UNAVAIL.
- listsnapshots: Controls whether snapshot information that is associated with this pool is displayed with the "zfs list" command. If this property is disabled, snapshot information can be displayed with the "zfs list -t snapshot" command. The default value is "off". Boolean value that can be either "off" or "on".
- readonly: Boolean value that can be either "off" or "on". Default value is "off". Controls setting the pool into read-only mode to prevent writes and/or data corruption.
- size: Read-only property that identifies the total size of the storage pool.
- version: Writable setting that identifies the current on-disk version of the pool. Can be any value from 1 to the output of the "zpool upgrade -v" command. This property can be used when a specific version is needed for backwards compatibility.

Getting and Setting Properties

There are a few ways you can get to the properties of your pool- you can get all properties at once, only one property, or more than one, comma-separated. For example, suppose I wanted to get just the health of the pool. I could issue the following command:

```
# zpool get health tank
NAME  PROPERTY  VALUE   SOURCE
tank  health    ONLINE  -
```

If I wanted to get multiple settings, say the health of the system, how much is free, and how much is allocated, I could issue this command instead:

```
# zpool get health,free,allocated tank
NAME  PROPERTY   VALUE   SOURCE
tank  health     ONLINE  -
tank  free       176G    -
tank  allocated  32.2G   -
```

And of course, if I wanted to get all the settings available, I could run:

```
# zpool get all tank
NAME  PROPERTY       VALUE       SOURCE
tank  size           208G        -
tank  capacity       15%         -
tank  altroot        -           default
tank  health         ONLINE      -
tank  guid           16951123779703469 70  default
tank  version        28          default
tank  bootfs         -           default
tank  delegation     on          default
tank  autoreplace    off         default
tank  cachefile      -           default
tank  failmode       wait        default
tank  listsnapshots  off         default
tank  autoexpand     off         default
tank  dedupditto     0           default
tank  dedupratio     1.00x       -
tank  free           176G        -
tank  allocated      32.2G       -
tank  readonly       off         -
tank  ashift         0           default
tank  comment        -           default
tank  expandsize     0           -
```

Setting a property is just as easy. However, there is a catch. For properties that require a string argument, there is no way to get it back to default. At least not that I am aware of. With the rest of the properties, if you try to set a property to an invalid argument, an error will print to the screen letting you know what is available, but it will not notify you as to what is default. However, you can look at the 'SOURCE' column. If the value in that column is "default", then it's default. If it's "local", then it was user-defined.

Suppose I wanted to change the "comment" property, this is how I would do it:

```
# zpool set comment="Contact admins@example.com" tank
# zpool get comment tank
NAME  PROPERTY  VALUE                       SOURCE
tank  comment   Contact admins@example.com  local
```

As you can see, the SOURCE is "local" for the "comment" property. Thus, it was user-defined. As mentioned, I don't know of a way to get string properties back to default after being set. Further, any modifiable property can be set at pool creation time by using the "-o" switch, as follows:

```
# zpool create -o ashift=12 tank raid1 sda sdb
```

Final Thoughts

The zpool properties apply to the entire pool, which means ZFS datasets will inherit that property from the pool. Some properties that you set on your ZFS dataset, which will be discussed towards the end of this series, apply to the whole pool. For example, if you enable block deduplication for a ZFS dataset, it dedupes blocks found in the entire pool, not just in your dataset. However, only blocks in that dataset will be actively deduped, while other ZFS datasets may not. Also, setting a property is not retroactive. In the case of your "autoexpand" zpool property to automatically expand the zpool size when all the drives have been replaced, if you replaced a drive before enabling the property, that drive will be considered a smaller drive, even if it physically isn't. Setting properties only applies to operations on the data moving forward, and never backward.

Despite a few of these caveats, having the ability to change some parameters of your pool to fit your needs as a GNU/Linux storage administrator gives you great control that other filesystems don't. And, as we've discovered thus far, everything can be handled with a single command "zpool", and easy-to-recall subcommands. We'll have one more post discussing a thorough examination of caveats that you will want to consider before creating your pools, then we will leave the zpool category, and work our way towards ZFS datasets, the bread and butter of ZFS as a whole. If there is anything additional about zpools you would like me to post on, let me know now, and I can squeeze it in.

Posted by Aaron Toponce on Wednesday, December 12, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 6 } Comments

1. Dominik Honnef | January 4, 2013 at 11:34 am | Permalink

   > I don’t know of a way to get string properties back to default after being set

   Indirectly, `zfs inherit` should be doing that, if you consider the inherited value the default. And if no parent specified it, it will truly be the default value.

2. Dominik Honnef | January 4, 2013 at 11:38 am | Permalink

   Disregard previous comment, obviously zfs != zpool…

3. Derek Scherger | April 13, 2014 at 5:48 pm | Permalink

   I think you can set a string property back to its default with something like this:

   zpool set bootfs= rpool

   This worked for me at least.

4. m.ardito | April 10, 2015 at 7:48 am | Permalink

   Hi, I'm learning zfs through your wonderful pages, thanks for sharing this!
   I have a doubt:

about ashift, you wrote "Default value is "9", as 2^9 = 512,"

but then when you use "#zpool get all tank" you get, amongst other settings,
"tank ashift 0 default"

and this is apparently in contrast to "However, you can look at the 'SOURCE' column. If the value in that column is "default", then it's default. If it's "local", then it was user-defined."

am I missing something?

5. Darael | April 17, 2015 at 3:47 pm | Permalink

   m.ardito - 0 is the default value, but what 0 means is "use the size the disk reports", and since a lot of AF disks report 512b sectors for compatibility with old OSes, 0 functionally means 9, even though this is suboptimal. To say the default is 9 is a simplification, but close enough.

6. MarkT | August 2, 2016 at 12:10 pm | Permalink

   A minor typo:
   "bootfs: Read-only..."

   man zpool:
   bootfs=pool/dataset
   listed under section describing:
   "... can be set at creation time and import time, and later changed with the zpool set command"

   Just though you would want to know.

{ 6 } Trackbacks

1. Aaron Toponce : ZFS Administration, Part II- RAIDZ | December 13, 2012 at 6:06 am | Permalink

   [...] Getting and Setting Properties [...]

2. Aaron Toponce : ZFS Administration, Part XII- Snapshots and Clones | January 7, 2013 at 9:19 pm | Permalink

   [...] Getting and Setting Properties [...]

3. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | April 19, 2013 at 4:56 am | Permalink

   [...] Getting and Setting Properties [...]

4. Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats | April 19, 2013 at 4:57 am | Permalink

   [...] Getting and Setting Properties [...]

5. Aaron Toponce : Install ZFS on Debian GNU/Linux | May 5, 2013 at 6:02 pm | Permalink

   [...] Getting and Setting Properties [...]

6. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | May 9, 2013 at 6:00 am | Permalink

   [...] Getting and Setting Properties [...]

Aaron Toponce

{ 2012.12.13 }

ZFS Administration, Part VIII- Zpool Best Practices and Caveats

Table of Contents

We now reach the end of ZFS storage pool administration, as this is the last post in that subtopic. After this, we move on to a few theoretical topics about ZFS that will lay the groundwork for ZFS Datasets. Our previous post covered the properties of a zpool. Without any ado, let's jump right into it. First, we'll discuss the best practices for a ZFS storage pool, then we'll discuss some of the caveats I think it's important to know before building your pool.

Best Practices

As with all recommendations, some of these guidelines carry a great amount of weight, while others might not. You may not even be able to follow them as rigidly as you would like. Regardless, you should be aware of them. I'll try to provide a reason why for each. They're listed in no specific order. The idea of "best practices" is to optimize space efficiency, performance and ensure maximum data integrity.

- Only run ZFS on 64-bit kernels. It has 64-bit specific code that 32-bit kernels cannot do anything with.
- Install ZFS only on a system with lots of RAM. 1 GB is a bare minimum, 2 GB is better, 4 GB would be preferred to start. Remember, ZFS will use 1/2 of the available RAM for the ARC.
- Use ECC RAM when possible for scrubbing data in registers and maintaining data consistency. The ARC is an actual read-only data cache of valuable data in RAM.
- Use whole disks rather than partitions. ZFS can make better use of the on-disk cache as a result. If you must use partitions, backup the partition table, and take care when reinstalling data into the other partitions, so you don't corrupt the data in your pool.
- Keep each VDEV in a storage pool the same size. If VDEVs vary in size, ZFS will favor the larger VDEV, which could lead to performance bottlenecks.

- Use redundancy when possible, as ZFS can and will want to correct data errors that exist in the pool. You cannot fix these errors if you do not have a redundant good copy elsewhere in the pool. Mirrors and RAID-Z levels accomplish this.
- ~~For the number of disks in the storage pool, use the "power of two plus parity" recommendation. This is for storage space efficiency and hitting the "sweet spot" in performance. So, for a RAIDZ-1 VDEV, use three (2+1), five (4+1), or nine (8+1) disks. For a RAIDZ-2 VDEV, use four (2+2), six (4+2), ten (8+2), or eighteen (16+2) disks. For a RAIDZ-3 VDEV, use five (2+3), seven (4+3), eleven (8+3), or nineteen (16+3) disks. For pools larger than this, consider striping across mirrored VDEVs.~~ **UPDATE:** The "power of two plus parity" is mostly a myth. See http://blog.delphix.com/matt/2014/06/06/zfs-stripe-width/ for more info.
- Consider using RAIDZ-2 or RAIDZ-3 over RAIDZ-1. You've heard the phrase "when it rains, it pours". This is true for disk failures. If a disk fails in a RAIDZ-1, and the hot spare is getting resilvered, until the data is fully copied, you cannot afford another disk failure during the resilver, or you will suffer data loss. With RAIDZ-2, you can suffer two disk failures, instead of one, increasing the probability you have fully resilvered the necessary data before the second, and even third disk fails.
- Perform regular (at least weekly) backups of the full storage pool. It's not a backup, unless you have multiple copies. Just because you have redundant disk, does not ensure live running data in the event of a power failure, hardware failure or disconnected cables.
- Use hot spares to quickly recover from a damaged device. Set the "autoreplace" property to on for the pool.
- Consider using a hybrid storage pool with fast SSDs or NVRAM drives. Using a fast SLOG and L2ARC can greatly improve performance.
- If using a hybrid storage pool with multiple devices, mirror the SLOG and stripe the L2ARC.
- If using a hybrid storage pool, and partitioning the fast SSD or NVRAM drive, unless you know you will need it, 1 GB is likely sufficient for your SLOG. Use the rest of the SSD or NVRAM drive for the L2ARC. The more storage for the L2ARC, the better.
- Keep pool capacity under 80% for best performance. Due to the copy-on-write nature of ZFS, the filesystem gets heavily fragmented. Email reports of capacity at least monthly.
- If possible, scrub consumer-grade SATA and SCSI disks weekly and enterprise-grade SAS and FC disks monthly. Depending on a lot factors, this might not be possible, so your mileage may vary. But, you should scrub as frequently as possible, basically.
- Email reports of the storage pool health weekly for redundant arrays, and bi-weekly for non-redundant arrays.
- When using advanced format disks that read and write data in 4 KB sectors, set the "ashift" value to 12 on pool creation for maximum performance. Default is 9 for 512-byte sectors.
- Set "autoexpand" to on, so you can expand the storage pool automatically after all disks in the pool have been replaced with larger ones. Default is off.
- Always export your storage pool when moving the disks from one physical system to another.
- When considering performance, know that for sequential writes, mirrors will always outperform RAID-Z levels. For sequential reads, RAID-Z levels will perform more slowly than mirrors on smaller data blocks and faster on larger data blocks. For random reads and writes, mirrors and RAID-Z seem to perform in similar manners. Striped mirrors will outperform mirrors and RAID-Z in both sequential, and random reads and writes.
- Compression is disabled by default. This doesn't make much sense with today's hardware. ZFS compression is extremely cheap, extremely fast, and barely adds any latency to the reads and writes. In fact, in some scenarios, your disks will respond faster with compression enabled than disabled. A further benefit is the massive space benefits.

Caveats

The point of the caveat list is by no means to discourage you from using ZFS. Instead, as a storage administrator planning out your ZFS storage server, these are things that you should be aware of, so as not to catch you with your pants down, and without your data. If you don't head these warnings, you could end up with corrupted data.

The line may be blurred with the "best practices" list above. I've tried making this list all about data corruption if not headed. Read and head the caveats, and you should be good.

- Your VDEVs determine the IOPS of the storage, and the slowest disk in that VDEV will determine the IOPS for the entire VDEV.
- ZFS uses 1/64 of the available raw storage for metadata. So, if you purchased a 1 TB drive, the actual raw size is 976 GiB. After ZFS uses it, you will have 961 GiB of available space. The "zfs list" command will show an accurate representation of your available storage. Plan your storage keeping this in mind.
- ZFS wants to control the whole block stack. It checksums, resilvers live data instead of full disks, self-heals corrupted blocks, and a number of other unique features. If using a RAID card, make sure to configure it as a true JBOD (or "passthrough mode"), so ZFS can control the disks. If you can't do this with your RAID card, don't use it. Best to use a real HBA.
- Do not use other volume management software beneath ZFS. ZFS will perform better, and ensure greater data integrity, if it has control of the whole block device stack. As such, avoid using dm-crypt, mdadm or LVM beneath ZFS.
- Do not share a SLOG or L2ARC DEVICE across pools. Each pool should have its own physical DEVICE, not logical drive, as is the case with some PCI-Express SSD cards. Use the full card for one pool, and a different physical card for another pool. If you share a physical device, you will create race conditions, and could end up with corrupted data.
- Do not share a single storage pool across different servers. ZFS is not a clustered filesystem. Use GlusterFS, Ceph, Lustre or some other clustered filesystem on top of the pool if you wish to have a shared storage backend.
- Other than a spare, SLOG and L2ARC in your hybrid pool, do not mix VDEVs in a single pool. If one VDEV is a mirror, all VDEVs should be mirrors. If one VDEV is a RAIDZ-1, all VDEVs should be RAIDZ-1. Unless of course, you know what you are doing, and are willing to accept the consequences. ZFS attempts to balance the data across VDEVs. Having a VDEV of a different redundancy can lead to performance issues and space efficiency concerns, and make it very difficult to recover in the event of a failure.
- Do not mix disk sizes or speeds in a single VDEV. Do mix fabrication dates, however, to prevent mass drive failure.
- In fact, do not mix disk sizes or speeds in your storage pool at all.
- Do not mix disk counts across VDEVs. If one VDEV uses 4 drives, all VDEVs should use 4 drives.
- Do not put all the drives from a single controller in one VDEV. Plan your storage, such that if a controller fails, it affects only the number of disks necessary to keep the data online.
- When using advanced format disks, you must set the ashift value to 12 at pool creation. It cannot be changed after the fact. Use "zpool create -o ashift=12 tank mirror sda sdb" as an example.
- Hot spare disks will not be added to the VDEV to replace a failed drive by default. You MUST enable this feature. Set the autoreplace feature to on. Use "zpool set autoreplace=on tank" as an example.
- The storage pool will not auto resize itself when all smaller drives in the pool have been replaced by larger ones. You MUST enable this feature, and you MUST enable it before replacing the first disk. Use "zpool set autoexpand=on tank" as an example.
- ZFS does not restripe data in a VDEV nor across multiple VDEVs. Typically, when adding a new device to a RAID array, the RAID controller will rebuild the data, by creating a new stripe width. This will free up some space on the drives in the pool, as it copies data to the new disk. ZFS has no such mechanism. Eventually, over time, the disks will balance out due to the writes, but even a scrub will not rebuild the stripe width.
- You cannot shrink a zpool, only grow it. This means you cannot remove VDEVs from a storage pool.
- You can only remove drives from mirrored VDEV using the "zpool detach" command. You can replace drives with another drive in RAIDZ and mirror VDEVs however.
- Do not create a storage pool of files or ZVOLs from an existing zpool. Race conditions will be present, and you will end up with corrupted data. Always keep multiple pools separate.
- The Linux kernel may not assign a drive the same drive letter at every boot. Thus, you should use the /dev/disk/by-id/ convention for your SLOG and L2ARC. If you don't, your zpool devices could end up as a SLOG device, which would in turn clobber your ZFS data.

- Don't create massive storage pools "just because you can". Even though ZFS can create 78-bit storage pool sizes, that doesn't mean you need to create one.
- Don't put production directly into the zpool. Use ZFS datasets instead.
- Don't commit production data to file VDEVs. Only use file VDEVs for testing scripts or learning the ins and outs of ZFS.

If there is anything I missed, or something needs to be corrected, feel free to add it in the comments below.

Posted by Aaron Toponce on Thursday, December 13, 2012, at 6:00 am. Filed under Debian, Linux, Ubuntu, ZFS. Follow any responses to this post with its comments RSS feed. You can post a comment or trackback from your blog. For IM, Email or Microblogs, here is the Shortlink.

{ 37 } Comments

1. boneidol | December 28, 2012 at 8:10 pm | Permalink

   "For the number of disks in the storage pool, use the “power of two plus parity” recommendation. This is for storage space efficiency and hitting the “sweet spot” in performance. So, for a RAIDZ-1 VDEV, use three (2+1), five (4+1), or nine (8+1) disks. For a RAIDZ-2 VDEV, use four (2+2), six (4+2), ten (8+2), or eighteen (16+2) disks. For a RAIDZ-3 VDEV, use five (2+3), seven (4+3), eleven (8+3), or nineteen (16+3) disks. For pools larger than this, consider striping across mirrored VDEVs."

   this differs from http://pthree.org/2012/12/05/zfs-administration-part-ii-raidz/ (and agrees with my math (and you instructions 🙂)

2. Aaron Toponce | December 29, 2012 at 6:24 am | Permalink

   Fixed the numbers in the post. Thanks!

3. Roger Hunwicks | January 12, 2013 at 11:00 am | Permalink

   When you say "zpool set auoresize=on tank”

   Do you really mean "zpool set autoexpand=on tank"

   I get an "invalid property" for both "set autoresize" and "set auoresize".

   Great series - thanks 🙂

4. Aaron Toponce | January 14, 2013 at 9:13 am | Permalink

   Yes. Typo. You know how when you get a word in your head, it seems to get applied to everything you type? Yeah. That happened here. Thanks for the notice. Fixed.

5. Dzezik | September 12, 2013 at 2:20 pm | Permalink

   raw size of 1TB drive is 931GiB
   -> 1TB is 10^12, GiB is 2^30
   -> (10^12)/(2^30)~=931

so
ZFS gives You 916,77GiB

6. Ghat Yadav | October 16, 2013 at 5:00 am | Permalink

   hi
   very nice and useful guide... however as a home user, I have a request for you to add one more section on how to add more drives to a existing pool.
   I started with a 4x4tb pool with raidz2. I have a 12bay device, and just populated 4 slots, for budget reasons, when I set it up I felt I will buy more disks in the future as they become cheap... but looks like thats not possible.
   once you create a zpool you cannot expand it (or am I wrong)...
   If I get 2 more 4TB disks, now if my budget allows, how do I best use them ?
   Ghat

7. JohnC | November 4, 2013 at 7:29 am | Permalink

   The Linux kernel may not assign a drive the same drive letter at every boot. Thus, you should use the /dev/disk/by-id/ convention for your SLOG and L2ARC. If you don't, your zpool devices "could end up as a SLOG device, which would in turn clobber your ZFS data."

   I think this has just happenned to me. I had a controller fail, after a series of reboots, I acquired a new controller. Now the disks on the new controller are fine, but the othed disks are "Faulted" with "corrupted data". I am sure the data is on them, but the order may be different. Loss of 8 of the 16 x 3Tb drives in a Raidz3 configuration is fatal.

   The status is Unavail with "the label is missing or invalid". How does one recover from this? Can it be done?

   Is there a fix for this

8. Aaron Toponce | November 5, 2013 at 6:40 pm | Permalink

   Restore from backup. That's probably the best you've got at this point. Corrupted data, and too many lost drives out of a RAID are very likely not recoverable.

9. Thumper advice | December 1, 2013 at 2:37 am | Permalink

   hi, I'm currently using a thumper (Sun X4500) and i'd like to give a try to ZFS on my SL64 x86_64. I'd like to export through NFS 22 x 1To hard drives. I know that there are a lot of options so basically I wanted to setup a RaidZ-1 with 21 hdd plus 1 spare drive. What do you think of that ? what about dedicating drives to ZIL and so on ?
   Thnks.
   François

10. Mark | March 12, 2014 at 4:56 am | Permalink

    "For the number of disks in the storage pool, use the "power of two plus parity" recommendation. "

    What is the rational behind this? I have 5x4TB drives, which given the drive size means that I should be using RAIDZ-2 but that doesn't fit within the guidelines. What kind of performance hit can I expect to take? Is it mainly a cpu bound issue?

    "Don't put production directly into the zpool. Use ZFS datasets instead. Don't commit production data to file VDEVs. Only use file VDEVs for testing scripts or learning the ins and outs of ZFS."

Can you elaborate on this? I don't quiet understand what you are saying in the above two points.

11. Mark Moorcroft | April 16, 2014 at 6:11 pm | Permalink

    Do you address SAS expanders with HBA's and Linux multipath devices anywhere here? My SAS drives appeared twice in /dev because the expander has multipath/failover . It wasn't clear if I should use mpath devices to build my zraid2 or what the best practice is in this case. I suspect this falls under "don't use mdadm/LVM", but I'm not sure. How do you get multipath load balancing with ZFS without using multipathd? My SAS drives ARE dual channel 512k block. LSI suggested just unplugging one of the cables to the backplane OR springing for their "new" hardware that they offer multipath support for. Of course I doubt that works with ZFS.

    --help ☹

12. Mark Moorcroft | April 16, 2014 at 6:33 pm | Permalink

    Follow-on question to my last: If using mpath to achieve load balancing/failover how would you go about getting ledctl or zfswatcher to work with mpath device names?

13. Sanjay | October 12, 2014 at 7:38 am | Permalink

    Aaron, kudos for this sweeping overview of zfs!

    >> Use whole disks rather than partitions. ZFS can make better use of the on-disk cache as a result.
    >> If you must use partitions, backup the partition table, and take care when reinstalling data into
    >> the other partitions, so you don't corrupt the data in your pool.

    Most recommenations (including yours) state that inability to use the on-disk cache the only reason for this recommendation. Are there any other reasons at all?

    I see more advantages from using slices than using a whole disk (especially todays large capacity disks).

    By carving each drive into n slices, an irecoverable error on any one slice will require a rebuild of only that slice and not the whole drive.

    Potentially, this can significantly reduce the rebuild time required to restore the zpool back to a healthy state, and based on the specific nature of the slice failure, one can then decide whether to preemptively migrate the other 7 slices as well, or continue using them a little longer.

    Of course, this advantage will not be available when there's a whole disk failure; but then, aren't partial failures more common than whole disk failures?

    The other advantage, from a data integrity point of view is, that I'm guaranteed that the on-disk cache cannot be used. (Given that the on-disk cache on HDDs is volatile, does zfs actually use this? And if it does,
    how is write integrity preserved?)

    I realise that this would necessitate an slog on a low latency non-volatile device, but when zfs is the choice based on data intregrity requirements, I don't see any other possibility that meets the requirement.

14. Mark | February 17, 2015 at 7:22 am | Permalink

    What do you mean with "ZFS does not restripe data in a VDEV"? I'm assuming here that if a disk gets broken and replaced, the resilvering process will rewrite the stripes to re-ensure redundancy, correct? I can understand when adding a new VDEV to a ZPOOL, that it may not start spreading the data out across the VDEVs. But since you can't grow a VDEV, I don't quite get the statement about intra-VDEV restriping.

Also, with modern home-NAS systems with SATA disks being around 6TB and higher, would a weekly scrub even be possible? A ZFS scrub takes longer than a MDADM-check, and that already takes me three days.

15. Aaron Toponce | February 17, 2015 at 1:29 pm | Permalink

> What do you mean with "ZFS does not restripe data in a VDEV"? I'm assuming here that if a disk gets broken and replaced, the resilvering process will rewrite the stripes to re-ensure redundancy, correct? I can understand when adding a new VDEV to a ZPOOL, that it may not start spreading the data out across the VDEVs. But since you can't grow a VDEV, I don't quite get the statement about intra-VDEV restriping.

What is meant, is that ZFS doesn't automatically rewrite data stripes when a new VDEV is added to the pool. For example, consider the following pool:

```
# zpool status pthree
  pool: pthree
 state: ONLINE
  scan: none requested
config:

        NAME              STATE     READ WRITE CKSUM
        pthree            ONLINE       0     0     0
          raidz1-0        ONLINE       0     0     0
            /tmp/file1    ONLINE       0     0     0
            /tmp/file2    ONLINE       0     0     0
            /tmp/file3    ONLINE       0     0     0
```

If you were to add another 3 disks in a RAIDZ1, thus creating a RAIDZ1+0, the newly created "raidz1-1" VDEV won't be automatically balanced with the data that resides on the "raidz1-0" VDEV:

```
zpool status pthree
  pool: pthree
 state: ONLINE
  scan: none requested
config:

        NAME              STATE     READ WRITE CKSUM
        pthree            ONLINE       0     0     0
          raidz1-0        ONLINE       0     0     0
            /tmp/file1    ONLINE       0     0     0
            /tmp/file2    ONLINE       0     0     0
            /tmp/file3    ONLINE       0     0     0
          raidz1-1        ONLINE       0     0     0
            /tmp/file4    ONLINE       0     0     0
            /tmp/file5    ONLINE       0     0     0
            /tmp/file6    ONLINE       0     0     0
```

ZFS will favor the "raidz1-1" VDEV for new writes, until the overall pool is balanced. And as data is modified, the data in the "raidz1-0" VDEV will eventually be balanced with the data on the "raidz1-1" VDEV, but that doesn't happen automatically, just because the new VDEV was added.

That is what is meant. I guess I could be clearer in that.

> Also, with modern home-NAS systems with SATA disks being around 6TB and higher, would a weekly scrub even be possible? A ZFS scrub takes longer than a MDADM-check, and that already takes me three days.

It depends entirely on how much data stored in the pool, how the pool is built, what type of drives make up the pool, and how busy the pool actually is. It may be possible, it may not. Your mileage will certainly

vary here. On some production servers, weekly scrubs work fine in 15x2TB pool arrays. On other storage servers, sometimes the scrub doesn't complete for a few weeks. So, I would say, it's "best practice", but it might not actually be doable, depending on some situations. I'll update the post.

16. John Naggets | February 27, 2015 at 7:06 am | Permalink

    Regarding the advanced format, my disks are 6 TB disks from Seagate (model: ST6000NM0034) and I checked they are using the 512e category of advanced format. Do you recommend me to set ashift=12 also for this category of disks?

17. Aaron Toponce | February 27, 2015 at 3:30 pm | Permalink

    I would. Some people swear by it that ashift=12 leads to substantially better performance, even when using the full disk. I personally haven't seen it, and I've worked with a good deal of AF disks. Shrug. It can't hurt, at least. So, I guess, why not?

18. John Naggets | February 28, 2015 at 11:01 am | Permalink

    Well I now tried out the ashift=12 option and yes I can also see a general performance gain of around 30% using simple "dd" read and write tests.

19. Aaron Toponce | March 3, 2015 at 11:51 am | Permalink

    Cool! Glad that works. I would put more through it with IOZone3 and Bonnie++, just to get a better representation of what the pool can do. But it sounds like "ashift=12" is the right option for your pool.

20. John Naggets | February 28, 2015 at 11:11 am | Permalink

    I have one remark though: I am using a 12 disks RAIDZ-2 pool with 6 TB disks which should give me in theory 60 TB of usable storage space, at the end I see only 49 TB of usable space, so somehow 11 TB get lost. Is this normal? I would expect some loss of space but 11 TB sounds quite a lot to me.

21. Aaron Toponce | March 3, 2015 at 12:01 pm | Permalink

    Hmm. 11 TB seems excessive. How are you getting to that data? The best way to see what is available versus what is used is with "zfs list". Add the "USED" and the "AVAIL" columns to get a better idea of what's available. Also, remember:

    > ZFS uses 1/64 of the available raw storage for metadata. So, if you purchased a 1 TB drive, the actual raw size is 976 GiB. After ZFS uses it, you will have 961 GiB of available space. The "zfs list" command will show an accurate representation of your available storage. Plan your storage keeping this in mind.

    So, first 6 TB according to the drive manufacturer is 6*1000^4 bytes. The actual raw size for the filesystem is 5.859 TiB. For a 12-disk RAIDZ-2, this means you'll have 10 * 5.859 TiB = 58.59 TiB. Then, 1/64 of that is used for metadata, so you have roughly 57.67 TiB usable. So, the question now remains, where is that other 8 TiB going? Not sure, but see what you get from summing the "USED" and "AVAIL" columns in "zfs list", and see where that puts you.

22. pdwalker | March 5, 2015 at 11:57 pm | Permalink

    Hi Aaron,

    Great guide. It's tremendously helpful.

One question though, under Caveats, the second last point is
Don't put production directly into the zpool. Use ZFS datasets instead.

Could you explain what that means? I'm afraid I don't follow what you mean by that.

23. Von Hawkins | March 16, 2015 at 1:51 pm | Permalink

    Minor edit.
    >>Email reports of the storage pool health weekly for redundant arrays, and bi-weekly for non-redundant arrays.

    It seems that you meant semi-weekly. Or am I reading this wrong. It makes sense to me that redundant arrays would need half the reporting frequency, but you stated that they need twice the frequency.

    Very helpful series. I am in the middle of purchasing a private cloud lab for home and plan to use FreeNAS. There is quite a lot to learn, but it seems most doable. --thanks

24. Magnus | March 29, 2015 at 7:47 am | Permalink

    Regarding 12*6 = 49 TB, well. 6 TB is actually 5.457 TiB (not 5.859). So that gives a total of 53.72 TiB usable space (with 1/64 removed). Which is "only" missing 4.72 TB (and quite close to 10%). I know linux does reserve 5% for root by default - I don't know if that translates to ZFS and/or whatever OS you are running, but might be worth looking into.

25. Xavier | July 2, 2015 at 3:31 am | Permalink

    * Don't trust df for monitoring, use zfs command (or zpool).
    * Be careful with snapshots because they can fill the pool (and df is not aware of this).

26. Ron Fish | November 6, 2015 at 11:26 am | Permalink

    Can a pool be spanned across multiple JBODs? In other words once i fill up my current pool am I dead ended and will have to create a new pool.

    I am using a ZFS pool for data storage in an environment that generates tons of data and the pool is nearly full with no more disk slots left in the array case.

27. Ed | November 26, 2015 at 7:09 am | Permalink

    >The storage pool will not auto resize itself when all smaller drives in the pool have been replaced by larger ones. You MUST enable this feature,

    100% true

    >and you MUST enable it before replacing the first disk. Use "zpool set autoexpand=on tank" as an example.

    It is easier to set autoexpand=on first then change to a larger drive, but it is not mandatory. If you forget (as I have done in the past), you can then 'zpool set autoexpand=on tank after-the-fact, then run 'zpool online -e tank devicename'.

28. Ed | November 26, 2015 at 7:24 am | Permalink

    >> Ron Fish wrote;
    >> Can a pool be spanned across multiple JBODs? In other words once i fill up my current pool am I dead ended and will have to create a new pool.

yes. zpools can be made up of anything you want it to be. USB drives, various hard disks, probably even floppy drives (shudder). Of course you are as fast as your slowest device, so it's better to be consistent. Just remember, if you 'zpool add', it's going to be there forever.

>>I am using a ZFS pool for data storage in an environment that generates tons of data and the pool is nearly full with no more disk slots left in the array case.

your options:
1) increase the size of the hard disks. You'll have to play games moving your data around to free a disk. Remember you cannot remove a disk (anything you did a zpool add on from a zpool, but you can remove a mirror (zpool attach).

2) if you are not using zfs compression, then you are sitting on a gold-mine of capacity. make a ZFS file system, enable compression and move your data from the non-compressed FS to the compressed. I had a fs with 1.1TB of data compress down to 300GB. As a bonus, all the nightly roll-up jobs making compressed archives could be turned off, leaving my historical data was immediately available.

I recommend reading the MAN page and look in the examples for zfs send | zfs receive. That's how you can pull your data from one zpool into another. Then you can play games with creating a file on NAS mounted storage, making it into a zpool, then transferring ZFS FS's out to free up space (think of it like archiving data)

29. Kai Harrekilde | February 8, 2016 at 7:04 am | Permalink

    Aaron,

    May I suggest to make a list of "Current Best Practice" with respect to attributes at pool creation time? I would add "compression=lz4" and "xattr=sa" to such a list along with ashift=12/13, autoexpand and autoreplace.

    It also seems that the L2ARC is a rather dubious win, according to /u/txgsync on https://www.reddit.com/r/zfs

30. John Mac | February 8, 2016 at 12:46 pm | Permalink

    The caveat/recommendation below is in most of the zfs zpool best practices guides, but can't find an explanation as to why. Information on why using mixed disk counts across vdevs is a bad practice is appreciated.

    "Do not mix disk counts across VDEVs. If one VDEV uses 4 drives, all VDEVs should use 4 drives."

31. Sebastian | March 8, 2016 at 2:29 am | Permalink

    Very nice articles all together, they have helped a a lot!!! Thanks

    I have one question.
    My setup is a box running proxmox using a RaidZ 10 setup using 4 USB 3 Sticks (each 32GB) as the root files system.
    For storage I used 6x 3TB WD Red in a second ZPool (RaidZ2). I will use different datasets in the storage pool to store all my different data. (VM disks, Movies, Personal Documents and so on...)
    My motherboard only has 6 SATA Ports (currently used by the 6 WD Red Disks). Thats why I went for the USB Stick install of proxmox (I working very well, no issues so far).

    Now I wanted to add Zil and L2ARC to my setup. Would I add both of them to both pools?? For me it makes sense that both pools should get a Zil and L2ARC to enhance performance.
    My Mainboard has one M.2 (Socket 3) and one additional SATA Express connector.

My idea was to purchase 2 SSDs (both 64 or 128GB) and connect them to the M.2 and SATA Express ports.
Then I would partition both to have 2 partitions with 1GB size and 2 partitions with the rest of the size shared (eg. 64GB SSD would partition into: sda1 (1GB) sda2 (1GB) sda3 (31GB) sda4 (31GB) in theory)
After that I could mirror the SLOG (Zil) over both devices for both pools and stripe the L2ARC for both pools over both SSDs.

Hopefully my explanation was clear.
Would that make sense, or is it bad to use the same SSD Device for two different pools?

32. Sebastian | March 8, 2016 at 2:34 am | Permalink

    Just to add something to my previous comment....
    I made a mistake about the SATA Express Connection on my motherboard, its not separate from the 6 SATA Ports, so I can't use it, since I need all 6 SATA Ports to connect the WD Red Disks.
    Can I use a similar setup with just one SSD in the M.2 Slot, and how would I partition that to have SLOG and L2ARC for two pools??

    Or should I look into buying a PCIe Card to get additional SATA ports for the second SSD device??

33. Brian Lachat | December 11, 2016 at 7:17 pm | Permalink

    First, Thanks so much for such a great write up. You state "Email reports of the storage pool health weekly for redundant arrays, and bi-weekly for non-redundant arrays." Perhaps I overlooked it but I don't see where It states how I can automate this. Would you please elaborate.

    Thanks,
    Brian

34. c0x | March 26, 2017 at 10:59 pm | Permalink

    ~# zpool list
    NAME SIZE ALLOC FREE EXPANDSZ FRAG CAP DEDUP HEALTH ALTROOT
    storage 31,9T 18,1T 13,8T - 16% 56% 1.00x ONLINE -
    zds 14,2T 6,85T 7,40T - 27% 48% 1.00x ONLINE -
    ftp 7,16T 5,66T 1,49T - 33% 79% 1.00x ONLINE -

    how to defrag this?

35. asmo | June 21, 2017 at 11:18 am | Permalink

    @ pdwalker

    I guess he ment that you can use /zpool when you created a pool without creating any datasets.

36. Martin Zuther | June 25, 2017 at 3:45 pm | Permalink

    Hi Aron,

    thanks for the great ZFS tutorial! I do have a question though. Where does the following recommendation come from?

    "Do not mix disk sizes [...] in a single VDEV. In fact, do not mix disk sizes [...] in your storage pool at all."

    You can find it all over the net, but there seems to be no one who ever explains it or points to the ZFS documentation. I'd like to exchange a 2 TB disk for a 3 TB one in a two-mirrored-disk setting (utilising the

"autoexpand" property) if that matters.

Martin

37. sherpa | September 4, 2018 at 4:08 am | Permalink

    what is recommended way of creating data pool if i have 13 x 5TB disks ? i could see in zol blogs that large disks should not be used with raidz but its not clear enough

{ 10 } Trackbacks

1. Links 16/12/2012: Humble Indie Bundle 7 Rants, ownCloud KDE Client | Techrights | December 15, 2012 at 7:30 pm | Permalink

   [...] ZFS Administration, Part VIII- Zpool Best Practices and Caveats [...]

2. Aaron Toponce : ZFS Administration, Part III- The ZFS Intent Log | December 20, 2012 at 8:08 am | Permalink

   [...] Best Practices and Caveats [...]

3. Aaron Toponce : ZFS Administration, Part I- VDEVs | December 20, 2012 at 8:09 am | Permalink

   [...] Best Practices and Caveats [...]

4. Aaron Toponce : ZFS Administration, Part II- RAIDZ | January 7, 2013 at 9:26 pm | Permalink

   [...] Best Practices and Caveats [...]

5. ZFS Homeserver Festplatten Beratung | January 31, 2013 at 6:47 am | Permalink

   [...] [...]

6. Aaron Toponce : ZFS Administration, Part X- Creating Filesystems | March 20, 2013 at 12:37 pm | Permalink

   [...] Best Practices and Caveats [...]

7. Aaron Toponce : ZFS Administration, Part XVII- Best Practices and Caveats | April 19, 2013 at 5:00 am | Permalink

   [...] Best Practices and Caveats [...]

8. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | July 2, 2013 at 7:24 am | Permalink

   […] Best Practices and Caveats […]

9. Aaron Toponce : ZFS Administration, Appendix B- Using USB Drives | July 8, 2013 at 10:07 pm | Permalink

   […] Best Practices and Caveats […]

10. Home Server (& Network) Setups - Page 4 | September 4, 2013 at 2:58 am | Permalink

[…] 8 drive raidz2s, which makes sense, but ZFS best practices according to this blog says different: Aaron Toponce : ZFS Administration, Part VIII- Zpool Best Practices and Caveats For the number of disks in the storage pool, use the “power of two plus parity” […]