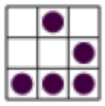

Aaron Toponce

{ 2013.04.19 }

ZFS Administration, Appendix A- Visualizing The ZFS Intent LOG (ZIL)

Table of Contents

Background

While taking a walk around the city with the rest of the system administration team at work today (we have our daily "admin walk"), a discussion came up about asynchronous writes and the contents of the ZFS Intent Log. Previously, as shown in the Table of Contents, I blogged about the ZIL in great length. However, I didn't really discuss what the contents of the ZIL were, and to be honest, I didn't fully understand it myself. Thanks to Andrew Kuhnhausen, this was clarified. So, based on the discussion we had during our walk, as well as some pretty graphs on the whiteboard, I'll give you the breakdown here.

Let's start at the beginning. ZFS behaves more like an ACID compliant RDBMS than a traditional filesystem. Its writes are transactions, meaning there are no partial writes, and they are fully atomic, meaning you get all or nothing. This is true whether the write is synchronous or asynchronous. So, best case is you have all of your data. Worst case is you missed the last transactional write, and your data is 5 seconds old (by default). So, let's look at those too cases- the synchronous write and the asynchronous write. With synchronous, we'll consider the write both with and without a separate logging device (SLOG).

The ZIL Function

The primary, and only function of the ZIL is to replay lost transactions in the event of a failure. When a power outage, crash, or other catastrophic failure occurs, pending transactions in RAM may have not been committed to slow platter disk. So, when the system recovers, the ZFS will notice the missing transactions. At this point, the ZIL is read to replay those transactions, and commit the data to stable storage. While the system is up and running, the ZIL is never read. It is only written to. You can verify this by doing the following (assuming you have SLOG in your system). Pull up two terminals. In one terminal, run an IOZone benchmark. Do something like the following:

```
$ iozone -ao
```

This will run a whole series of tests to see how your disks perform. While this benchmark is running, in the other terminal, as root, run the following command:

```
# zpool iostat -v 1
```

This will clearly show you that when the ZIL resides on a SLOG, the SLOG devices are only written to. You never see any numbers in the read columns. This is becaus the ZIL is never read, unless the need to replay transactions from a crash are necessary. Here is one of those seconds illustrating the write:

```
                                                    capacity     operations    bandwidth
pool                                              alloc   free   read  write   read  write
------------------------------------------------  -----  -----  -----  -----  -----  -----
pool                                              87.7G   126G      0    155      0   601K
  mirror                                          87.7G   126G      0    138      0   397K
    scsi-SATA_WDC_WD2500AAKX-_WD-WCAYU9421741-part5      -      -      0     69      0   727K
    scsi-SATA_WDC_WD2500AAKX-_WD-WCAYU9755779-part5      -      -      0     68      0   727K
logs                                                 -      -      -      -      -      -
  mirror                                          2.43M   478M      0      8      0   108K
    scsi-SATA_OCZ-REVODRIVE_XOCZ-6G9S9B5XDR534931-part1      -      -      0      8      0   108K
    scsi-SATA_OCZ-REVODRIVE_XOCZ-THM0SU3H89T5XGR1-part1      -      -      0      8      0   108K
  mirror                                          2.57M   477M      0      7      0  95.9K
    scsi-SATA_OCZ-REVODRIVE_XOCZ-V402GS0LRN721LK5-part1      -      -      0      7      0  95.9K
    scsi-SATA_OCZ-REVODRIVE_XOCZ-WI4ZOY2555CH3239-part1      -      -      0      7      0  95.9K
cache                                                -      -      -      -      -      -
  scsi-SATA_OCZ-REVODRIVE_XOCZ-6G9S9B5XDR534931-part5    26.6G  56.7G      0      0      0      0
  scsi-SATA_OCZ-REVODRIVE_XOCZ-THM0SU3H89T5XGR1-part5    26.5G  56.8G      0      0      0      0
  scsi-SATA_OCZ-REVODRIVE_XOCZ-V402GS0LRN721LK5-part5    26.7G  56.7G      0      0      0      0
  scsi-SATA_OCZ-REVODRIVE_XOCZ-WI4ZOY2555CH3239-part5    26.7G  56.7G      0      0      0      0
------------------------------------------------  -----  -----  -----  -----  -----  -----
```

The ZIL should always be on non-volatile stable storage! You want your data to remain consistent across power outages. Putting your ZIL on a SLOG that is built from TMPFS, RAMFS, or RAM drives that are not battery backed means you will lose any pending transactions. This doesn't mean you'll have corrupted data. It only means you'll have old data. With the ZIL on volatile storage, you'll never be able to get the new data that was pending a write to stable storage. Depending on how busy your servers are, this could be a Big Deal. SSDs, such as from Intel or OCZ, are good cheap ways to have a fast, low latentcy SLOG that is reliable when power is cut.

Synchronous Writes without a SLOG

When you do not have a SLOG, the application only interfaces with RAM and slow platter disk. As previously discussed, the ZFS Intent LOG (ZIL) can be thought of as a file that resides on the slow platter disk. When the application needs to make a synchronous write, the contents of that write are sent to RAM, where the application is currently living, as well as sent to the ZIL. So, the data blocks of your synchronous write at this exact moment in time have two homes- RAM and the ZIL. Once the data has been written to the ZIL, the platter disk sends an acknowledgement back to the application letting it know that it has the data, at which point the data is flushed from RAM to slow platter disk.

This isn't ALWAYS the case, however. In the case of slow platter disk, ZFS can actually store the transaction group (TXG) on platter immediately, with pointers in the ZIL to the locations on platter. When the disk ACKs back that the ZIL contains the pointers to the data, then the write TXG is closed in RAM, and the space in the ZIL opened up for future transactions. So, in essence, you could think of the TXG SYNCHRONOUS write commit happening in three ways:

1. All data blocks are synchronously written to both the RAM ARC and the ZIL.
2. All data blocks are synchronously written to both the RAM ARC and the VDEV, with pointers to the blocks written in the ZIL.
3. All data blocks are synchronously written to disk, where the ZIL is completely ignored.

In the image below, I tried to capture a simplified view of the first process. The pink arrows, labeled as number one, show the application committing its data to both RAM and the ZIL. Technically, the application is running in RAM already, but I took it out to make the image a bit more clean. After the blocks have been committed to RAM, the platter ACKs the write to the ZIL, noted by the green arrow labeled as number two. Finally, ZFS flushes the data blocks out of RAM to disk as noted by the gray arrow labeled as number three.

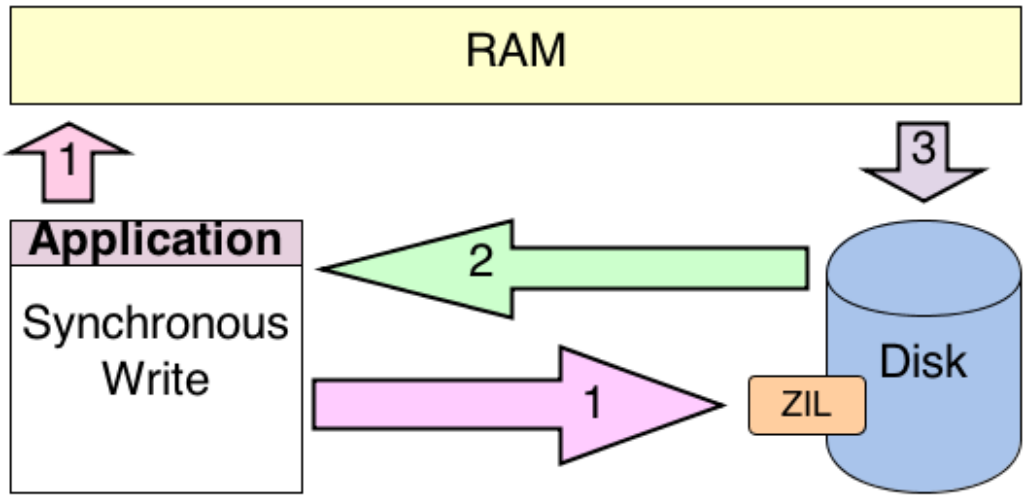

Image showing a synchronous write with ZFS without a SLOG

Synchronous Writes with a SLOG

The advantage of a SLOG, as previously outlined, is the ability to use low latency, fast disk to send the ACK back to the application. Notice that the ZIL now resides on the SLOG, and no longer resides on platter. The SLOG will catch all synchronous writes (well those called with O_SYNC and fsync(2) at least). Just as with platter disk, the ZIL will contain the data blocks the application is trying to commit to stable storage. However, the SLOG, being a fast SSD or NVRAM drive, ACKs the write to the ZIL, at which point ZFS flushes the data out of RAM to slow platter.

Notice that ZFS is not flushing the data out of the ZIL to platter. This is what confused me at first. The data is flushed from RAM to platter. Just like an ACID compliant RDBMS, the ZIL is only there to replay the transaction, should a failure occur, and the data is lost. Otherwise, the data is never read from the ZIL. So really, the write operation doesn't change at all. Only the location of the ZIL changes. Otherwise, the operation is exactly the same.

As shown in the image, again the pink arrows labeled number one show the application committing its data to both the RAM and the ZIL on the SLOG. The SLOG ACKs the write, as identified by the green arrow labeled number two, then ZFS flushes the data out of RAM to platter as identified by the gray arrow labeled number three.

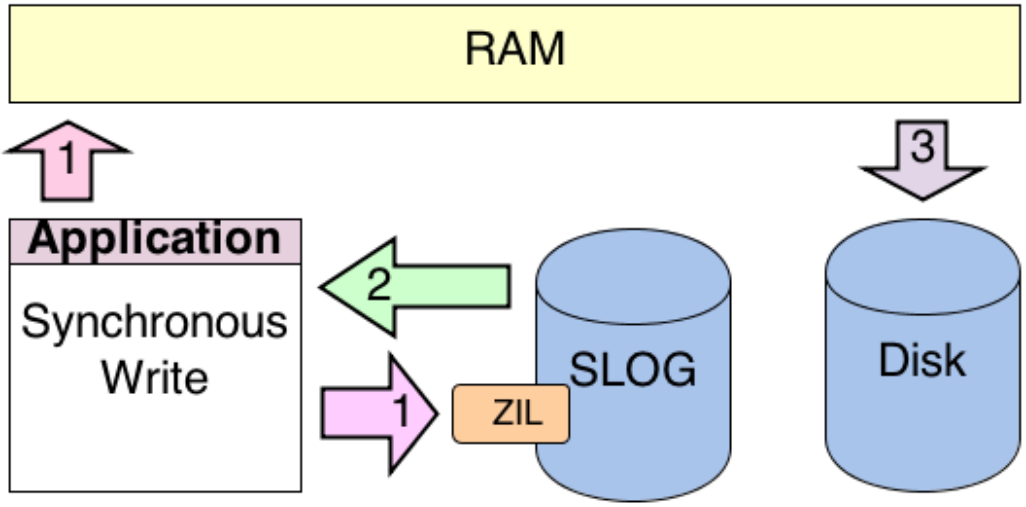

Image showing a synchronous write with ZFS with a SLOG

Asynchronous Writes

Asynchronous writes have a history of being "unstable". You have been taught that you should avoid asynchronous writes, and if you decide to go down that path, you should prepare for corrupted data in the event of a failure. For most filesystems, there is good counsel there. However, with ZFS, it's a nothing to be afraid of. Because of the architectural design of ZFS, all data is committed to disk in transaction groups. Further, the transactions are atomic, meaning you get it all, or you get none. You never get partial writes. This is true with asynchronous writes. So, your data is ALWAYS consistent on disk- even with asynchronous writes.

So, if that's the case, then what exactly is going on? Well, there actually resides a ZIL in RAM when you enable "sync=disabled" on your dataset. As is standard with the previous synchronous architectures, the data blocks of the application are sent to a ZIL located in RAM. As soon as the data is in the ZIL, RAM acknowledges the write, and then flushes the data do disk, as would be standard with synchronous data.

I know what you're thinking: "Now wait a minute! The are no acknowledgements with asynchronous writes!" Not always true. With ZFS, there is most certainly an acknowledgement, it's just one coming from very, very fast and extremely low latent volatile storage. The ACK is near instantaneous. Should there be a crash or some other failure that causes RAM to lose power, and the write was not saved to non-volatile storage, then the write is lost. However, all this means is you lost new data, and you're stuck with old but **consistent** data. Remember, with ZFS, data is committed in atomic transactions.

The image below illustrates an asynchronous write. Again, the pink number one arrow shows the application data blocks being initially written to the ZIL in RAM. RAM ACKs back with the green number two arrow. ZFS then flushes the data to disk, as per every previous implementation, as noted by the gray number 3 arrow. Notice in this image, even if you have a SLOG, with asynchronous writes, it's bypassed, and never used.

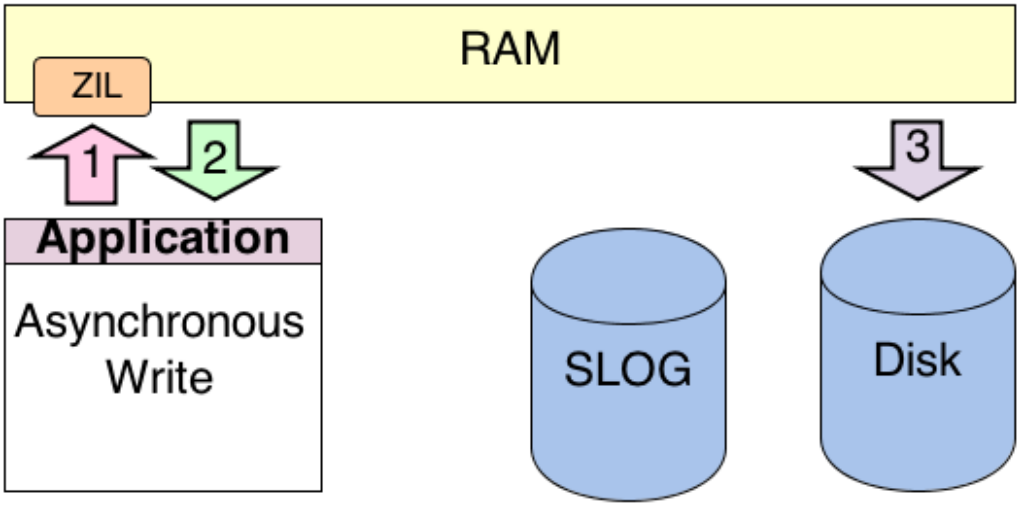

Image showing an asynchronous write with ZFS.

Disclaimer

This is how I and my coworkers understand the ZIL. This is after reading loads of documentation, understanding a bit of computer science theory, and understanding how an ACID compliant RDBMS works, which is architected in a similar manner. If you think this is not correct, please let me know in the comments, and we can have a discussion about the architecture.

There are certainly some details I am glossing over, such as how much data the ZIL will hold before its no longer utilized, timing of the transaction group writes, and other things. However, it should also be noted that aside from some obscure documentation, there doesn't seem to be any solid examples of exactly how the ZIL functions. So, I thought it would be best to illustrate that here, so others aren't left confused like I was. For me, images always make things clearer to understand.

Posted by Aaron Toponce on Friday, April 19, 2013, at 5:00 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.

{ 5 } Comments

1. udc | April 22, 2013 at 6:30 pm | Permalink

   Hi, first of all thanks for the whole ZFS series, you write in an easily understandable way and your practical experiences and opinions gives an additional value to it.
   (Though, not trying to be a pedant or anything but sometimes you present a personal opinion as a solid unchangeable fact, like in the deduplication section where you ended up in conclusion that you need x multiplies RAM of what seems really necessary because of some hard-written ratio. I didn't check the source code whether there is really such a ratio but even if so this is an open source software so nothing is set in stone and thus nothing is stopping you or anyone else to simply go and change it, thus saying that to effectively manage deduplication for a 12 TB storage you need not 60 GB, letting the actual reasoning behind this first number itself aside, but actually 240 GB RAM, that's kind of cheap. Don't take it in a wrong way.)

   What provoked me to a reaction is this appendix article. It seems to me that the images are drawn in a little bit unfortunate way. Let's take the first image. You made it look like the application when writing to ZFS without a separate ZIL writes its data first to the ZIL and also to the RAM, then it gets confirmation of write done and

then the data is written from RAM to the disk. That's exactly what your picture and arrows show. But the application is already in the RAM and all its data is thus already in the RAM, also the ZIL is already the disk, so what you are actually indicating here is that the data is pointlessly duplicated first in the memory while being simultaneously written to the disk and then it's again pointlessly duplicated, only this time in the disk! I didn't check the source code but honestly I very much doubt that the ZFS developers would be that stupid. I would accept that the data is duplicated in the RAM, or more precisely put it's copied to the ZFS part of memory, but I will certainly not accept that the data is written to the slow disk twice.
Also, you kind of make the ZIL look like some special entity that resides in the disk but is somewhat not part of the disk and you are somewhat hinting that the data is written to the ZIL (on the disk) and then it's duplicated or moved from the ZIL to the disk, effectively writing the same data to the same disk twice. A non-IT analogy to this idea would be a secretary in a company who sits at her front desk and when a messenger from the outside world comes in and brings a package she receives it, logs (records) this event and then stands up and goes back inside the company to deliver the package to the actual recipient. Surely that's not what is happening. If that idea was true then all journaling file systems would have twice as slow writing speed as the non-journaling file systems. ZIL is nothing special, it's an ordinary journal. The journaling is not about the middle man through whom the data is passed to the disk (thus if the journal is actually residing on the disk the data is written to the disk twice), it's about the way how the data is written to the disk, i.e. in a documented way without making premature assumption that the write will be finished successfully and thus without "touching" any live data or file system structures that's already on the disk, all that to make sure that in case of a failure resulting in an unfinished write there would be no, or very little, inconsistency.
And lastly, your remark that after the write done is acknowledged to the application the data is flushed to the disk (letting alone the fact that the data was on the disk already safe and sound) goes against the whole principle of a synchronous write. The point of synchronous write is that the writing application is not notified about the write done until the data is really written to the target. You of course do know that, it's just a little bit unfortunate wording you have chosen.

So I would suggest you to consider the following adjustment to the first picture. Make the application box a little part of the RAM rectangle, pretty much in a similar way as the little ZIL rectangle is a part of the cylinder. Then draw another little box next to the application box within the RAM rectangle and call it "ZFS". Then the whole process would be: a very short horizontal arrow "1" between application and ZFS boxes within the RAM rectangle, then vertical arrow "2" between ZFS box in the RAM and the ZIL box in the disk, and lastly again a short horizontal arrow "3" back between ZFS and application boxes within the RAM. Plain and simple.

The next picture would be similarly (1) application to ZFS, (2) ZFS to ZIL (in SLOG), (3) ZFS back to application, (4) ZFS to disk. As a side effect this would also remove any confusion you noticed you have had as for whether there is write from ZIL to the disk in (4). Of course not, of course the write goes from the ZFS box that's (as any other process) obviously in the memory.

The last picture would be (1) application to ZFS, (2) ZFS to ZIL (in RAM, thus there would be 3 little boxes as part of the RAM rectangle), (3) ZFS to application, (4) ZFS to disk. Although thinking about it maybe even more precise would be to draw the ZIL box in the RAM as a part (a sub-box, if you will) of the ZFS box, thus the arrows would be: (1) application to ZFS (containing ZIL inside), (2) ZFS to application, (3) ZFS to disk. It's simplified but that's what pictures are for, to show clearly and simply the process.

Just my 2 cents.

2. Aaron Toponce | April 23, 2013 at 7:33 am | Permalink

> But the application is already in the RAM and all its data is thus already in the RAM,

I mentioned that in the post. I said:

> When the application needs to make a synchronous write, the contents of that write are sent to RAM, where the application is currently living, as well as sent to the ZIL.

Then in the next paragraph, I said:

> Technically, the application is running in RAM already, but I took it out to make the image a bit more clean.

With regards to the ZIL, you should read the zil.c source code. This is exactly the behavior. The data is indeed written twice, but as I mentioned in the article, I am glossing over some details, such as when the ZIL is used and when it isn't with respect to data size. In some cases, the ZIL is completely bypassed, even for fully synchronous writes. In some cases, it stores the pointers to the TXG on slower platter disk, rather than the data itself. And then in some cases, which my blog addresses, it stores the actual data blocks. It's highly variable on the environment, the intense details of which, are not relevant to this post. You should read https://blogs.oracle.com/realneel/entry/the_zfs_intent_log if you want to get a better understanding of when it is used, and when it isn't. I'll update the post to make mention of the pointers on slower platter disk though. Thanks for helping me realize I missed that.

> ZIL is nothing special, it’s an ordinary journal.

Heh. No, it's very special. The function of the ZIL is to replay the last transaction in the event of a catastrophe. Without a fsck(8). This is not the function of a journal. With a filesystem journal, the journal is opened before the write, the write occurs, for both the inodes then the data blocks, then the journal is closed. For every one application write, 4 disk writes are committed, one after the other. If a power outage occurs while committing data to the inode, but before committing the blocks, you have data corruption. A fsck(8) will identify the opened idnode, and read whatever data is in that physical location on disk, but it may or may not be the data you're looking for, the latter likely the case.

With the ZIL, the data is committed to the ZIL by ZFS, which could be on slow platter, as a single write. Then when the disk ACKs the write, a second write is committed to disk with the same transaction. This transaction group (TXG) includes the checksums, inodes, data blocks, etc. All in one write. So, if there is a power outage, either you got the write, or you didn't (part of the atomic nature of the transaction). So, you either have new data, or old data. But not corrupted data. Also, we only have 2 writes to disk, instead of 4, as you would with a journal. You really should read how transactional databases works, because ZFS is very, very similar in function.

> And lastly, your remark that after the write done is acknowledged to the application the data is flushed to the disk (letting alone the fact that the data was on the disk already safe and sound) goes against the whole principle of a synchronous write. The point of synchronous write is that the writing application is not notified about the write done until the data is really written to the target. You of course do know that, it’s just a little bit unfortunate wording you have chosen.

This is EXACTLY the function of the ZIL. When the ZIL has the data, it's on stable storage. Thus, we can ACK back to the application the data has been committed to stable storage. When the ZIL in on a fast SSD or RAM disk, the ACK is substantially improved. The application can move on to other functions faster as a result. This is documented all over the Web. I appreciate your concern for my knowledge on the subject, but you REALLY should read the docs. Your lack of understanding how transactional writes work is showing your ignorance.

> So I would suggest you to consider the following adjustment to the first picture. Make the application box a little part of the RAM rectangle, pretty much in a similar way as the little ZIL rectangle is a part of the cylinder. Then draw another little box next to the application box within the RAM rectangle and call it “ZFS”. Then the whole process would be: a very short horizontal arrow “1″ between application and ZFS boxes within the RAM rectangle, then vertical arrow “2″ between ZFS box in the RAM and the ZIL box in the disk, and lastly again a short horizontal arrow “3″ back between ZFS and application boxes within the RAM. Plain and simple.

I chose the design I did, because it's easy to understand. And, if you read the post, you'll see where I made clarifications, such as the application already residing in RAM. Making those adjustments would complicate the picture and add a lot of unnecessary noise. If you read the post, you understand what the images are communicating.

Thanks for stopping by. Please also read the following documentation about the ZIL, transaction groups, and just ZFS in general:

* Official docs: http://docs.oracle.com/cd/E19253-01/819-5461/
* Async writes: http://www.racktopsystems.com/dedicated-zfs-intent-log-aka-slogzil-and-data-fragmentation/
* FAQ about the ZIL and SSDs: http://constantin.glez.de/blog/2011/02/frequently-asked-questions-about-flash-memory-ssds-and-zfs#benefit. Also read http://constantin.glez.de/blog/2011/02/frequently-asked-questions-about-flash-memory-ssds-and-zfs#spacezil about the space and what it contains.
* And probably the best post on the subject, which confirms everything in my post: http://nex7.blogspot.com/2013/04/zfs-intent-log.html

> So, to summarize so far, you've got a ZFS Intent Log that is very similar to the log a SQL database uses, write once and forget (unless something bad happens), and you've got an in-RAM write cache and transaction group commits that handle actually writing to the data vdevs (and by the by, the txg commits are sequential, so all your random write traffic that came in between commits is sequential when it hits disk). The write cache is volatile as its in RAM, so the ZIL is in place to store the synchronous writes on stable media to restore from if things go south.

I think I understand the ZIL. Please read the docs.

3. patrick domack | April 27, 2013 at 12:54 pm | Permalink

   yes, sync writes are always written twice.
   once to the intent log (journal) and once to the pool.

   if you have no slog (seperate log/journal), then the journal exists within your pool.
   and if your paranoud, ext4 will work this same method by adjusting its options.

   i didnt see any talk about the performance issues with using a slog though.

   all writes to an slog happen one at a time (queue depth=1 always). so the latency of your writes matter all other performance metrics on you slog device.

   if you have multible slog devs you can increase performance, cause each will be used for a different writer, but only multible writers cause async blocks the current one.

   while zfs is waiting for the slog, it will group other writes together to create a larger transaction though, so all isnt lost, but you generally are going have to have a lot of writers to gain performance here.

   other things to muse over,
   zfs never verifies writes to the slog are correct, till it needs them for recovery.
   zfs depends on the slog not to lie about it flushing data to disk, or you might as well use async writes instead.

4. Richard Elling | May 10, 2014 at 12:06 pm | Permalink

   Each dataset has a separate ZIL and there is some concurrency related to multithreaded sync writes for interesting workloads (think databases or NFS). Thus the "ZIL" is often "ZILs" and the workload is only single-threaded in the degenerate case. Expecting queue depth =1 is a bad assumption.

5. Roger Qiu | June 24, 2014 at 2:34 am | Permalink

   Asynchronous writes seems to have a higher performance than synchronous writes.

   However this article http://milek.blogspot.com.au/2010/05/zfs-synchronous-vs-asynchronous-io.html says that asynchronous writes could result in inconsistency from the application's point of view. And it especially affects databases, since databases may be implementing their own transactions.

   In what situations would asynchronous writes be preferable to synchronous writes?

{ 1 } Trackback

1. Aaron Toponce : ZFS Administration, Part XIII- Sending and Receiving Filesystems | July 2, 2013 at 7:25 am | Permalink

   […] Visualizing The ZFS Intent Log (ZIL) […]

Aaron Toponce

{ 2013.05.09 }

ZFS Administration, Appendix B- Using USB Drives

Table of Contents

Introduction

This comes from the "why didn't I think of this before?!" department. I have lying around my home and office a ton of USB 2.0 thumb drives. I have six 16GB drives and eight 8GB drives. So, 14 drives in total. I have two hypervisors in a GlusterFS storage cluster, and I just happen to have two USB squids, that support 7 USB drives each. Perfect! So, why not put these to good use, and add them as L2ARC devices to my pool?

Disclaimer

USB 2.0 is limited to 40 MBps per controller. A standard 7200 RPM hard drive can do 100 MBps. So, adding USB 2.0 drives to your pool as a cache is not going to increase the read bandwidth. At least not for large sequential reads. However, the seek latency of a NAND flash device is typically around 1 milliseconds to 3 milliseconds, whereas a platter HDD is around 12 milliseconds. If you do a lot of small random IO, like I do, then your USB drives will actually provide an overall performance increase that HDDs cannot provide.

Also, because there are no moving parts with NAND flash, this is less data that needs to be read from the HDD, which means less movement of the actuator arm, which means consuming less power in the long term. So, not only are they better for small random IO, they're saving you power at the same time! Yay for going green!

Lastly, the L2ARC *should* be read intensive. However, it can also be write intensive if you don't have enough room in your ARC and L2ARC to store all the requested data. If this is the case, you'll be constantly writing to your L2ARC. For USB drives without wear leveling algorithms, you'll chew through the drive quickly, and it will be dead in no time. If this is your case, you could store only metadata, rather than the actual data block pages in the L2ARC. You can do this with the following:

```
# zfs set secondarycache=metadata pool
```

You can set this pool-wide, or per dataset. In the case outlined above, I would certainly do it pool-wide, which each dataset will inherit by default.

Implementation

To this up, it's rather straight forward. Just identify what the drives are, by using their unique identifiers, then add them to the pool:

```
# ls /dev/disk/by-id/usb-* | grep -v part
/dev/disk/by-id/usb-Kingston_DataTraveler_G3_0014780D8CEBEBC145E80163-0:0@
/dev/disk/by-id/usb-Kingston_DataTraveler_SE9_00187D0F567FEC2090007621-0:0@
/dev/disk/by-id/usb-Kingston_DataTraveler_SE9_00248121ABD5EC2070002E70-0:0@
/dev/disk/by-id/usb-Kingston_DataTraveler_SE9_00D0C9CE66A2EC2070002F04-0:0@
/dev/disk/by-id/usb-_USB_DISK_Pro_070B2605FA99D033-0:0@
/dev/disk/by-id/usb-_USB_DISK_Pro_070B2607A029C562-0:0@
/dev/disk/by-id/usb-_USB_DISK_Pro_070B2608976BFD58-0:0@
```

So, there are my seven drives that I outlined at the beginning of the post. So, to add them to the system as L2ARC drives, just run the following command:

```
# zpool add -f pool cache usb-Kingston_DataTraveler_G3_0014780D8CEBEBC145E80163-0:0\
usb-Kingston_DataTraveler_SE9_00187D0F567FEC2090007621-0:0\
usb-Kingston_DataTraveler_SE9_00248121ABD5EC2070002E70-0:0\
usb-Kingston_DataTraveler_SE9_00D0C9CE66A2EC2070002F04-0:0\
usb-_USB_DISK_Pro_070B2605FA99D033-0:0\
usb-_USB_DISK_Pro_070B2607A029C562-0:0\
usb-_USB_DISK_Pro_070B2608976BFD58-0:0
```

Of course, these are the unique identifiers for my USB drives. Change them as necessary for your drives. Now that they are installed, are they filling up?

```
# zpool iostat -v
pool                                                        alloc   free   read  write   read  write
----------------------------------------------------------  -----  -----  -----  -----  -----  -----
pool                                                         695G  1.13T     21     59  53.6K   457K
  mirror                                                     349G   579G     10     28  25.2K   220K
    ata-ST1000DM003-9YN162_S1D1TM4J                             -      -      4     21  25.8K   267K
    ata-WDC_WD10EARS-00Y5B1_WD-WMAV50708780                     -      -      4     21  27.9K   267K
  mirror                                                     347G   581G     11     30  28.3K   237K
    ata-WDC_WD10EARS-00Y5B1_WD-WMAV50713154                     -      -      4     22  16.7K   238K
    ata-WDC_WD10EARS-00Y5B1_WD-WMAV50710024                     -      -      4     22  19.4K   238K
logs                                                            -      -      -      -      -      -
  mirror                                                     4K  1016M      0      0      0      0
    ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part1                -      -      0      0      0      0
    ata-OCZ-REVODRIVE_OCZ-X5RG0EIY7MN7676K-part1                -      -      0      0      0      0
cache                                                           -      -      -      -      -      -
  ata-OCZ-REVODRIVE_OCZ-33W9WE11E9X73Y41-part2              52.2G    16M      4      2  51.3K   291K
  ata-OCZ-REVODRIVE_OCZ-X5RG0EIY7MN7676K-part2              52.2G    16M      4      2  52.6K   293K
  usb-Kingston_DataTraveler_G3_0014780D8CEBEBC145E80163-0:0   465M  6.80G      0      0    319  72.8K
  usb-Kingston_DataTraveler_SE9_00187D0F567FEC2090007621-0:0  1.02G  13.5G      0      0  1.58K  63.0K
  usb-Kingston_DataTraveler_SE9_00248121ABD5EC2070002E70-0:0  1.17G  13.4G      0      0    844  72.3K
  usb-Kingston_DataTraveler_SE9_00D0C9CE66A2EC2070002F04-0:0   990M  13.6G      0      0  1.02K  59.9K
  usb-_USB_DISK_Pro_070B2605FA99D033-0:0                    1.08G  6.36G      0      0  1.18K  67.0K
  usb-_USB_DISK_Pro_070B2607A029C562-0:0                    1.76G  5.68G      0      1  2.48K   109K
  usb-_USB_DISK_Pro_070B2608976BFD58-0:0                    1.20G  6.24G      0      0    530  38.8K
----------------------------------------------------------  -----  -----  -----  -----  -----  -----
```

Something important to understand here, is the drives do not need to be all the same size. You can mix and match as you have on hand. Of course, the more space you can give to the cache, the better off you'll be.

Conclusion

While this certainly isn't designed for speed, it can be used for lower random IO latencies, and it well reduce power in the datacenter. Further, what else are you going to do with those USB devices just lying around? Might as well put them to good use. Definitely seeing as though "the cloud" is making it trivial to get all of your files online.

Posted by Aaron Toponce on Thursday, May 9, 2013, 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. Jeremy Rosengren | May 9, 2013 at 8:03 am | Permalink

   Question: It looks like you already have a couple of OCZ RevoDrives installed... in this particular scenario, do the USB cache devices still provide value? It does look like they're being used, does ZFS treat all the disks as the same speed, and if so, couldn't that hurt performance by having cache data written to devices that are slower than your SSDs?

2. Aaron Toponce | May 9, 2013 at 8:44 am | Permalink

   Yes and no. First, ZFS is smart enough to know that the OCZ drives are faster than the USB sticks, so it will favor putting data there before using the USB drives. However, Having the USB drives will mean decreased seek latencies in retrieving data that would normally be on platter. So, it certainly doesn't hurt the pool at all, even if the USB sticks can't retrieve the data as quickly as the OCZ drives. But you are right that a cached page that once lived on the OCZ drives that now resides on the USB drives, will be accessed slower than before. But it's still faster than pulling it off platter for small random IO.

3. Anonymous | May 29, 2017 at 12:34 pm | Permalink

   Are you using a USB 2.0 Hub on a USB 2.0 port?
   What about using a USB 3 Hub on a USB 3 port? (and what about 3.0 versus 3.1 Gen2).

   I mean, about using a lot of ols USB 2.0 Sticks on a HUB that is USB 3.x, would it pass beyond 40MB/s!

   And what about IOPS (number of operations per second).

   My tests (with EXT4) with more than five hundred old USB sticks give an impressive 9.5Gigabits/s.

4. Paranoin. Green Powe | May 29, 2017 at 12:44 pm | Permalink

   You say if saves energy! I am not sure of that.

   Some USB sticks consume a lot of power, like Sandisk USB 3 64GiB, after five minutes of written a full virtual disk image backup (more than 10Gigabytes) it is so hot you can not touch it or you get burn.

   So moving the head is less power than power used by such USB stick.

   Anonymous: How much power uses your old (>500) usb sticks plus hubs? I understand you to not buy a SSD (no one give such great speed yet), but can you pay so much electricity bill (speculating on power)?

   I had measure power for such Sandisk with a device that is set in middle, it drains more than 20 watts when writting at full USB 3 (5gb/s) speed... ihave no USB 3.1 Gen 2 (10gp/s) ports

Aaron Toponce

{ 2013.12.10 }

ZFS Administration, Appendix C- Why You Should Use ECC RAM

Table of Contents

Introduction

With the proliferation of ZFS into FreeBSD, Linux, FreeNAS, Illumos, and many other operating systems, and with the introduction of OpenZFS to unify all the projects under one collective whole, more and more people are beginning to tinker with ZFS in many different situations. Some install it on their main production servers, others install it on large back-end storage arrays, and even yet, some install it on their workstations or laptops. As ZFS grows in popularity, you'll see more and more ZFS installations on commodity hardware, rather than enterprise hardware. As such, you'll see more and more installations of ZFS on hardware that does not support ECC RAM.

The question I pose here: Is this a bad idea? If you spend some time searching the Web, you'll find posts over and over on why you should choose ECC RAM for your ZFS install, with great arguments, and for good reason too. In this post, I wish to reiterate those points, and make the case for ECC RAM. Your chain is only as strong as your weakest link, and if that link is non-ECC RAM, you lose everything ZFS developers have worked so hard to achieve on keeping your data from corruption.

Good RAM vs Bad RAM vs ECC RAM

To begin, let's make a clear distinction between "Good RAM" and "Bad RAM" and how that compares to "ECC RAM":

- Good RAM- High quality RAM modules with a low failure rate.
- Bad RAM- Low quality RAM modules with a high failure rate.
- ECC RAM- RAM modules with error correcting capabilities.

"Bad RAM" isn't necessarily non-ECC RAM. I've deployed bad ECC RAM in the past, where even though they are error correcting, they fail frequently, and need to be replaced. Further, ECC RAM isn't necessarily "Good RAM". I've deployed non-ECC RAM that has been in production for years, and have yet to see a corrupted file due to not having error correction in the hardware. The point is, you can have exceptional non-ECC "Good RAM" that will never fail you, and you can have horrid ECC "Bad RAM" that still creates data corruption.

What you need to realize is the *rate of failure*. An ECC RAM module can fail just as frequently as a non-ECC module of the same build quality. Hopefully, the failure rate is such that ECC can fix the errors it detects, and still function without data corruption. But just to beat a dead horse dead, ECC RAM and hardware failure rates are disjointed. Just because it's ECC RAM does not mean that the hardware fails less frequently. All it means is that it detects the failures, and attempts to correct them.

ECC RAM

Failure rates are hard to get a handle on. If you read the Wikipedia article on ECC RAM, it mentions a couple studies that have been attempted to get a handle on how often bit errors occur in DIMM modules:

> Work published between 2007 and 2009 showed widely varying error rates with over 7 orders of magnitude difference, ranging from 10^(−10) [to] 10^(−17) error/bit-h[ours], roughly one bit error, per hour, per gigabyte of memory to one bit error, per millennium, per gigabyte of memory. A very large-scale study based on Google's very large number of servers was presented at the SIGMETRICS/Performance'09 conference. The actual error rate found was several orders of magnitude higher than previous small-scale or laboratory studies, with 25,000 to 70,000 errors per billion device hours per megabit (about 2.5^(−7) x 10^(−11) error/bit-h[ours])(i.e. about 5 single bit errors in 8 Gigabytes of RAM per hour using the top-end error rate), and more than 8% of DIMM memory modules affected by errors per year.

So roughly, from what Google was seeing in their datacenters, 5 bit errors in 8 GB of RAM per hour in 8% of their installed RAM. If you don't think this is significant, you're fooling yourself. Most of these bit errors are caused by background radiation affecting the installed DIMMs, due to neutrons from cosmic rays. But voltage fluctuations, bad circuitry, and just poor build quality can also come in as factors to "bit flips" in your RAM.

ECC RAM works by detecting this bad bit by using an extra parity bit per byte. In other words, for every 8 bits, there is a 9th parity bit which operates as the checksum for the previous 8. So, for a DIMM module registering itself as 64 GB to the system, there is actually 72 GB physically installed on the chip to give space for parity. However, it's important to note that ECC RAM can only correct 1 bit flip per byte (8 bits). If you have 2 bit flips per byte, ECC RAM will not be able to recover the data.

ZFS was designed to detect silent data errors that happen due to hardware and other factors. ZFS checksums your data from top to bottom to ensure that you do not have data corruption. If you've read this series from the beginning, you'll know how ZFS is architected, and how data integrity is first priority for ZFS. People who use ZFS use it because they cannot stand data corruption anywhere in their filesystem, at any time. However, if your RAM is not ECC RAM, then you do not have the guarantee that your file is not corrupt when stored to disk. If the file was corrupted in RAM, due to a frozen bit in RAM, then when stored to ZFS, it will get checksummed with this bad bit, as ZFS will assume the data it is receiving is good. As such, you'll have corrupted data in your ZFS dataset, and it will be checksummed corrupted, with no way to fix the error internally.

This is bad.

A scenario

To drive the point home further about ECC RAM in ZFS, let's create a scenario. Let's suppose that you are not using ECC RAM. Maybe this is installed on your workstation or laptop, because you like the ZFS userspace

tools, and you like the idea behind ZFS. So, you want to use it locally. However, let's assume that you have non-ECC "Bad RAM" as defined above. For whatever reason, you have a "frozen bit" in one of your modules. The DIMM is only storing a "0" or a "1" in a specific location. Let's say it always reports a "0" due to the hardware failure, no matter what should be written there. To keep things simple, we'll look at 8 bits, or 1 byte in our example. I'll show the bad bit with a red "0".

Your application wishes to write "11001011", but due to your Bad RAM, you end up with "11000011". As a result, "11000011" is sent to ZFS to be stored. ZFS adds a checksum to "11000011" and stores it in the pool. You have data corruption, and ZFS doesn't know any different. ZFS assumes that the data coming out of RAM is intentional, so parity and checksums are calculated based on that result.

But what happens when you read the data off disk and store it back in your faulty non-ECC RAM? Things get ugly at this point. So, you read back "11000011" to RAM. However, it's stored in _almost_ the same position before it was sent to disk. Assume it is stored only 4 bits later. Then, you get back "01000011". Not only was your file corrupt on disk, but you've made things worse by storing them back into RAM where the faulty hardware is. But, ZFS is designed to correct this, right? So, we can fix the bad bit back to "**1**1000011", but the problem is that the data *is still corrupted*!

Things go downhill from here. Because this is a physical hardware failure, we actually can't set that first bit to "1". So, any attempt at doing so, will immediately revert it back to "0". So, while the data is stored in our faulty non-ECC RAM, the byte will remain as "01000011". Now, suppose we're ready to flush the data in RAM to disk, we've compounded our errors by storing "01000011" on platter. ZFS calculates a new checksum based on the newly corrupted data, again assuming our DIMM modules are telling us the truth, and we've further corrupted our data.

As you can see, the more we read and write data to and from non-ECC RAM, the more we have a chance of corrupting data on the filesystem. ZFS was designed to protect us against this, but our chain is only as strong as the weakest link, which in this case is non-ECC RAM corrupting our data.

You might think that backups can save you. If you have a non-corrupted file you can restore from, great. However, if your ZFS snapshot, or rsync(1) copied over the corrupted bit to your backups, then you're sunk. And ZFS scrubbing won't help you here either. As already mentioned, you stored corrupted data on disk *correctly*, meaning the checksum for the corrupted byte is correct. However, if you have additional bad bits in RAM, scrubbing will actually try to "fix" the bad bits. But, because it's a hardware failure, causing a frozen bit, the scrub will continue, and continue, thrashing your pool. So, scrubbing will actually bring performance to its knees, trying to fix that one bad bit in RAM. Further, scrubbing won't fix bad bits already stored in your pool that have been properly checksummed.

No matter how you slice and dice it, you trusted your non-ECC RAM, and your trusty RAM failed you, with no recourse to fall back on.

ECC Pricing

Due to the extra hardware on the DIMM, ECC RAM is certainly more costly than their non-ECC counterparts, but not by much. In fact, because ECC DIMMs have 9/8 additional more hardware, the price pretty closely reflects that. In my experience, 64 GB of ECC RAM is roughly 9/8 more costly than 64 GB of non-ECC RAM. Many general purpose motherboards will support unbuffered ECC RAM also, although you should choose a motherboard that supports active ECC scrubbing, to keep bit corruption minimized.

You can get high quality ECC DDR3 SDRAM off of Newegg for about $50 per 4 GB. Non-ECC DDR3 SDRAM retails for almost exactly the same price. To me, it just seems obvious. All you need is a motherboard supporting it, and Supermicro motherboards supporting ECC RAM can also be relatively inexpensive. I know this is subjective, but I recently built a two-node KVM hypervisor shared storage cluster with 32 GB of registered ECC RAM in each box with Tyan motherboards. Total for all 32 GB was certainly more costly than

everything else in the system, but I was able to get them at ~ $150 per 16 GB, or $600 for all 64 GB total. The boards were ~$250 each, or $500 total for two boards. So, it total, for two very beefy servers, I spent ~$1100, minus CPU, disk, etc. To me, this is a small investment to ensure data integrity, and I would not have saved much going the non-ECC route.

The very small extra investment was very much worth it, to make sure I have data integrity from top-to-bottom.

Conclusion

ZFS was built from the ground up with parity, mirroring, checksums and other mechanisms to protect your data. If a checksum fails, ZFS can make attempts at loading good data based on redundancy in the pool, and fix the corrupted bit. But ZFS is assuming that a correct checksum means the bits were correct before the checksum was applied. This is where ECC RAM is so critical. ECC RAM can greatly reduce the risk that your bits are not correct before they get stored into the pool.

So, some lessons you should take away from this article:

- ZFS checksums assume the data is correct from RAM.
- Regular ZFS scrubs will greatly reduce the risk of corrupted bits, but can be your worst enemy with non-ECC RAM hardware failures.
- Backups are only as good as the data they store. If the backup is corrupted, it's not a backup.
- ZFS parity data, checksums, and physical data all need to match. When they don't, repairs start taking place. If it is corrupted out the gate, due to non-ECC RAM, why are you using ZFS again?

Thanks to "cyberjock" on the FreeBSD forums for inspiring this post.

Posted by Aaron Toponce on Tuesday, December 10, 2013, at 9:40 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.

{ 14 } Comments

1. Anonymous | December 10, 2013 at 7:46 pm | Permalink

   While the price for ECC RAM is reasonable, Intel forces you into server class motherboards and Xeon chips to be able to use ECC RAM. Fortunately an AMD Phenom combined with an Asus motherboard will work with ECC RAM. Yes it is slower and less energy efficient, but the price to performance ratio is much better. This is what I am using for my home ZFS on Linux file server.

2. Chris | December 11, 2013 at 4:44 am | Permalink

   first of all, thanks for this great series of posts!
   I have a question regarding unbuffered-vs-registered DIMMs: "although you should choose a motherboard that supports active ECC scrubbing, to keep bit corruption minimized, which would require registered ECC DIMMs."
   are you referring to patrol vs demand scrubbing as described here http://en.wikipedia.org/wiki/Memory_scrubbing#Scrubbing_Types ? Only regECC would then allow for patrol scrubbing. Or am I missing something?

3. Aaron Toponce | December 11, 2013 at 10:19 am | Permalink

   ECC must be supported by the CPU. AMD has broad support for ECC in a lot of their chips, but for Intel, this means the Xeons only. So, your CPU choice will limit your motherboard choice. Some motherboard BIOS settings allow you to enable active ECC scrubbing, while others allow you to set the scrub frequency. However, some motherboards don't have any such ECC scrub support in the BIOS.

   I was always under the impression that ECC scrubbing support in the BIOS required buffered, or registered ECC RAM. However, after doing a bit of research, that appears to not be the case. As such, I've updated the post.

4. Ivar | April 27, 2014 at 7:13 am | Permalink

   Thanks for the informative post. I just have a nitpick on the description of ECC memory:

   "ECC RAM works by detecting this bad bit by using an extra parity bit per byte. In other words, for every 8 bits, there is a 9th parity bit which operates as the checksum for the previous 8.[...] However, it's important to note that ECC RAM can only correct 1 bit flip per byte (8 bits). If you have 2 bit flips per byte, ECC RAM will not be able to recover the data."

   To me this makes it sound like it is possible to correct a single-bit error in a byte using only a single parity bit. This is of course impossible. ECC RAM generally uses 8 bits per 64 bits, and can then correct a single bit of error in those 64 bits, or detect (but not correct) two bits of error.

5. Philip Robar | June 11, 2014 at 1:00 am | Permalink

   > AMD has broad support for ECC in a lot of their chips, but for Intel, this means the Xeons only.

   This is not true. Many Intel desktop CPUs of recent generations support ECC memory: 11 of 30 non-legacy Celerons, 21 of 38 non-legacy Pentiums and all 19 3rd and 4th generation (Ivy Bridge and Haswell respectively) Core i3s support ECC memory. Also there are Xeons that do not support ECC memory and all Atom Processors for Storage and Servers do support ECC memory. (All other Atoms don't.)

   Intel's Ark site has very has extensive filtering capabilities that let you find just the right processor for your needs: http://ark.intel.com

6. Cédric Dufour | January 30, 2015 at 10:13 am | Permalink

   Thank you for this excellent ZFS serie!

   This particular post had me baffled and alarmed with the huge DIMM error rate reported ("5 bit errors per 8GB per hour"; wow!).

   And I couldn't get the meaning of "more than 8% of DIMM memory modules affected by errors per year".

   So I went through the Google/Sigmetrics09 article.
   Worth noting in this article:

   [ch. 3.2] "the [...] number of correctable errors per year is highly variable [perhaps] because the majority of the DIMMs see zero errors, while those affected see a large number of them".

   [ch. 5.4] "corectable error rates starts to increase quickly as the [DIMM] population ages beyond 10 months up until around 20 months, [after which] the correctable error incidence remains constant. [...] this may indicate that older DIMMs that did not have correctable errors in the past, possibly will not develop them later on"

[ch.7] "over 8% of DIMMs [...] saw at least one correctable error per year"

[ch.7] "error rates are unlikely to be dominated by soft errors" (as opposed to hard errors; hard errors = hardware defect = reproducible errors)

Those are no excuses for not using ECC for mission-critical applications.

But it can certainly help (and ease the alarm) of those who can not go ECC.

7. Aaron Toponce | February 17, 2015 at 1:48 pm | Permalink

> Thank you for this excellent ZFS serie!

No problem. Glad you're enjoying it!

> This particular post had me baffled and alarmed with the huge DIMM error rate reported ("5 bit errors per 8GB per hour"; wow!).
>
> And I couldn't get the meaning of "more than 8% of DIMM memory modules affected by errors per year".
>
> So I went through the Google/Sigmetrics09 article.
> Worth noting in this article:
>
> [ch. 3.2] "the [...] number of correctable errors per year is highly variable [perhaps] because the majority of the DIMMs see zero errors, while those affected see a large number of them".
>
> [ch. 5.4] "corectable error rates starts to increase quickly as the [DIMM] population ages beyond 10 months up until around 20 months, [after which] the correctable error incidence remains constant. [...] this may indicate that older DIMMs that did not have correctable errors in the past, possibly will not develop them later on"
>
> [ch.7] "over 8% of DIMMs [...] saw at least one correctable error per year"
>
> [ch.7] "error rates are unlikely to be dominated by soft errors" (as opposed to hard errors; hard errors = hardware defect = reproducible errors)
>
> Those are no excuses for not using ECC for mission-critical applications.
>
> But it can certainly help (and ease the alarm) of those who can not go ECC.

Agreed. ZFS isn't unique with ECC RAM. ECC RAM should be deployed whenever fiscally possible, ZFS or not. But, when deploying ZFS with non-ECC RAM, you lose the guarantee that ZFS will keep your data in tact and correct. But, that's the case with any filesystem. Again, ZFS isn't unique here.

8. wondra | May 9, 2015 at 1:17 am | Permalink

I agree with Ivar. It is impossible to correct errors with only parity. Also, the configuration 8+1 bits was last used before memory modules were introduced, when there were 64 megabit x1 chips installed in single sockets on the mainboard. Nowadays, the memory lines are much longer and that governs the width of the ECC code that must be used.
en.m.wikipedia.org/wiki/Hamming_distance
en.m.wikipedia.org/wiki/ECC_memory
en.m.wikipedia.org/wiki/Hamming_code

9. Scott S. | February 5, 2016 at 6:06 pm | Permalink

For what it is worth you may as well buy and use ECC memory. The interesting thing to note is with Solaris on FMA (fault management architecture) if a scenario such as a bit is constantly flipping on some exact memory cell then the page will be disabled, so you can use the ECC memory to even greater effect than any other OS to my knowledge (As I've not seen it implemented in any version of Linux yet nor BSD's). Over time more cells may (or will) fail so can allow you more time before you must buy a replacement or find out that some manufactures memory modules have more faults than another's. Just my little add-on here.

10. Yatti420 | February 21, 2016 at 3:05 am | Permalink

    Look it's 2016! Just get the ECC RAM!! It's not like it's costing an arm and a leg for consumer builds..

11. Michael | February 29, 2016 at 10:10 am | Permalink

    Here is an interesting article explaining that ZFS does not corrupt your data even if your RAM goes south. Thus, bad ECC dimms will not corrupt your data:
    http://jrs-s.net/2015/02/03/will-zfs-and-non-ecc-ram-kill-your-data/

    "...Say you only corrupt one block in 5,000 this way. That would still be hellacious. So let’s examine the more reasonable idea of corrupting some data due to bad RAM during a scrub. And let’s assume that we have RAM that not only isn’t working 100% properly, but is actively goddamn evil and trying its naive but enthusiastic best to specifically kill your data during a scrub:

    First, you read a block. This block is good. It is perfectly good data written to a perfectly good disk with a perfectly matching checksum. But that block is read into evil RAM, and the evil RAM flips some bits. Perhaps those bits are in the data itself, or perhaps those bits are in the checksum. Either way, your perfectly good block now does not appear to match its checksum, and since we’re scrubbing, ZFS will attempt to actually repair the “bad” block on disk. Uh-oh! What now?

    Next, you read a copy of the same block – this copy might be a redundant copy, or it might be reconstructed from parity, depending on your topology. The redundant copy is easy to visualize – you literally stored another copy of the block on another disk. Now, if your evil RAM leaves this block alone, ZFS will see that the second copy matches its checksum, and so it will overwrite the first block with the same data it had originally – no data was lost here, just a few wasted disk cycles. OK. But what if your evil RAM flips a bit in the second copy? Since it doesn’t match the checksum either, ZFS doesn’t overwrite anything. It logs an unrecoverable data error for that block, and leaves both copies untouched on disk. No data has been corrupted. A later scrub will attempt to read all copies of that block and validate them just as though the error had never happened, and if this time either copy passes, the error will be cleared and the block will be marked valid again (with any copies that don’t pass validation being overwritten from the one that did)..."

    Also, Matt Ahrens (one of the ZFS architects) explains that ECC RAM is not needed. Also, he says that ZFS can checksum the data in RAM to catch errors in ECC dimms:
    http://arstechnica.com/civis/viewtopic.php?f=2&t=1235679&p=26303271#p26303271
    "...There's nothing special about ZFS that requires/encourages the use of ECC RAM more so than any other filesystem. If you use UFS, EXT, NTFS, btrfs, etc without ECC RAM, you are just as much at risk as if you used ZFS without ECC RAM. Actually, ZFS can mitigate this risk to some degree if you enable the unsupported ZFS_DEBUG_MODIFY flag (zfs_flags=0x10). This will checksum the data while at rest in memory, and verify it before writing to disk, thus reducing the window of vulnerability from a memory error.

    I would simply say: if you love your data, use ECC RAM. Additionally, use a filesystem that checksums your data, such as ZFS..."

12. Daryl | June 16, 2017 at 11:01 am | Permalink

DDR4 supposedly improves error handling, with CRC checks and on-chip parity detection, over DDR3. How does this stack up in comparison with ECC?

13. Klaus | August 28, 2017 at 10:21 am | Permalink

    @Daryl: The first DDR4 modules on the market had ECC. Non-ECC-DDR4-RAM appeared later on the market. That probably explains the (false) rumor that "DDR4 has better error handling than DDR3". Plus, there are numerous articles on the web which "prove" the increased reliability of DDR4-RAM (with ECC) by comparing it to DDR3-RAM...without ECC. Yep. Very funny.

    I do not yet know how DDR4 compares to DDR3 regarding reliability. However, we do know that DDR3 was more reliable than DDR2-RAM. The Google report to which the article refers showed high error rates in DDR2-RAM. Note that at this time Google also did not replace RAM which began to show correctable errors - no wonder you see higher error rates when you decide to keep your failing RAM in use. Also note that Google used non-standard memory modules which were, according to the specs, incompatible with the mainboards (they worked in real life, of course, but possibly less reliably than standard modules).

    Back to DDR4: DDR4-RAM can *optionally* have a "Write CRC" feature which can detect errors occurring on the bus when data is written to the RAM (the host could then retry the data transmission). However, this optional feature will, AFAIK, not be present on non-ECC-DDR4-RAM.

14. Marvin Glenn | January 29, 2018 at 11:20 pm | Permalink

    Please see comment #4 by Ivar and let me echo his sentiment. When the extra bit is only taken as a parity bit for a byte, you can only detect a single bit error, but not correct it. ECC looks at a piece of data larger than a byte and considers it against more than one extra bit. From that, it can detect and often properly correct errors in memory. But calling it 'parity' should be avoided as 'parity' is only really an "error detection code", not an "error correction code".

Aaron Toponce

{ 2013.12.18 }

ZFS Administration, Appendix D- The True Cost Of Deduplication

Table of Contents

This post gets filed under the "budget and planning" part of systems administration. When planning out your ZFS storage pool, you will need to make decision about space efficiency, and the cost required to build out that architecture. We've heard over and over that ZFS block deduplication is expensive, and I've even mentioned it on this blog, but how expensive is it really? What are we looking at out of pocket? That's what this post is about. We'll look at it from two perspectives- enterprise hardware and commodity hardware. We should be able to make some decent conclusions after looking into it.

We're only going to address storage, not total cost which would include interconnects, board, CPU, etc. Those costs can be so variable, it can make this post rather complicated. So, let's stick with the basics. We're going to define enterprise hardware as 15k SAS drives and SLC SSDs, and we'll define commodity hardware as 7200 SATA drives and MLC SSDs. In both cases, we'll stick with high quality ECC DDR3 RAM modules. We'll use a base ZFS pool of 10TB.

So, without further ado, let's begin.

Determining Disk Needs

Before we go off purchasing hardware, we'll need to know what we're looking at for deduplication, and if it's a good fit for our data needs. This can be a hard puzzle to solve, without actually storing all the data in the pool, and seeing when you end up. However, here are a few ideas for coming to that solution (the "three S tests"):

1. **Sample Test**: Get a good representative sample of your data. You don't need a lot. Maybe 1/5 of the full data. Just something that represents what will actually be stored. This will be the most accurate test,

provided you get a good sample- the more, the better. Store that sample on the deduplicated pool, and see where you end up with your dedupratio.

2. **Simulation Test**: This will be less accurate than the sample test above, but it can still give a good idea of what you'll be looking at. Run the "zfs -S" command, and see where the cards fall. This will take some time, and may stress your pool, so run this command off hours, if you must do it on a production pool. It won't actually deduplicate your data, just simulate it. Here is actual an actual simulation histogram from my personal ZFS production servers:

```
# zdb -S
Simulated DDT histogram:

bucket              allocated                       referenced
______   ______________________________   ______________________________
refcnt   blocks   LSIZE   PSIZE   DSIZE   blocks   LSIZE   PSIZE   DSIZE
------   ------   -----   -----   -----   ------   -----   -----   -----
     1    5.23M    629G    484G    486G    5.23M    629G    484G    486G
     2     860K   97.4G   86.3G   86.6G    1.85M    215G    190G    190G
     4    47.6K   4.18G   3.03G   3.05G     227K   19.7G   14.2G   14.3G
     8    11.8K    931M    496M    504M     109K   8.49G   4.25G   4.33G
    16    3.89K    306M   64.3M   68.3M    81.8K   6.64G   1.29G   1.37G
    32    5.85K    499M    116M    122M     238K   17.9G   4.64G   4.86G
    64    1.28K   43.7M   20.0M   21.0M     115K   3.74G   1.69G   1.79G
   128    2.60K   50.2M   20.0M   22.0M     501K   9.22G   3.62G   3.99G
   256      526   6.61M   3.18M   3.62M     163K   1.94G    946M   1.06G
   512      265   3.25M   2.02M   2.19M     203K   2.67G   1.72G   1.86G
    1K      134   1.41M    628K    720K     185K   2.13G    912M   1.02G
    2K       75   1.16M    188K    244K     222K   3.37G    550M    716M
    4K       51    127K   85.5K    125K     254K    657M    450M    650M
    8K        2      1K      1K   2.46K    26.7K   13.3M   13.3M   32.8M
   16K        1     512     512   1.94K    31.3K   15.6M   15.6M   60.7M
 Total    6.15M    732G    574G    576G    9.38M    920G    708G    712G

dedup = 1.24, compress = 1.30, copies = 1.01, dedup * compress / copies = 1.60
```

3. **Supposed Test**: Basically, just guess. It's by far the least accurate of our testing, but you might understand your data better than you think. For example, is this 10TB server going to be a Debian or RPM package repository? If so, the data is likely highly duplicated, and you could probably get close to 3:1 savings, or better. Maybe this server will store a lot of virtual machine images, in which case the base operating system will be greatly duplicated. Again, your ratios could be very high as a result. But, you know what you are planning on storing, and what to expect.

Now you'll have a deduplication ratio number. In my case, it's 1.24:1. This number will help us "oversubscribe" our storage. In order to determine how much disk to purchase, our equation should be:

```
Savings = Need - (Need / Ratio)
```

With my ratio of 1.24:1, which is running about a dozen virtual machines, rather than purchasing the full 10TB of disk, we really only need to purchase 8TB of disk. This is a realistic expectation. So, I can save purchasing 2TB worth of storage for this setup. The question then becomes whether or not those savings are worth it.

Determining RAM Needs

Ok, now that we know how much disk to purchase, we now need to determine how much RAM to purchase. Already, we know that the deduplication table (DDT) will occupy no more than 25% of installed RAM, by default. This is adjustable with the kernel module, but we'll stick with default for this post. So, we just need to determine how large that 25% is, so we can understand exactly how much RAM will be needed to safely store the ARC without spilling over to spinning platter disk. In order to get a handle on this metric, we have two options:

1. **Counting Blocks**: With the "zdb -b" command, you can count the number of currently used blocks in your pool. As with the "zdb -S" command, this will stress your pool, but it will give you the most accurate picture of what to expect with a deduplication table. Below is an actual counting of block on my production servers:

```
# zdb -b pool

Traversing all blocks to verify nothing leaked ...

        No leaks (block sum matches space maps exactly)

        bp count:        11975124
        bp logical:    1023913523200      avg:  85503
        bp physical:    765382441472      avg:  63914     compression:   1.34
        bp allocated:   780946764288      avg:  65214     compression:   1.31
        bp deduped:             0    ref>1:      0   deduplication:   1.00
        SPA allocated: 780946764288     used: 39.19%
```

   In this case, I have 11975124 used blocks, and my 2 TB pool is 39.19% full, or 784GB. Thus, each block is about 70KB in size. You might see something different. According to Oracle, each deduplicated block will occupy about 320 bytes in RAM. Thus, 2TB divided by 70KB blocks gives a total storage space of about 30,700,000 total blocks. 30,700,000 blocks multiplied by 320 bytes, is 9,824,000,000 bytes, or 9.8GB of RAM for the DDT. Because the DDT is no more than 25% of ARC, and the ARC is typically 25% of RAM, I need at least 156.8GB, or basically 160GB of installed RAM to prevent the DDT from spilling to spinning platter disk.

2. **Rule of Thumb**: This is our "rule of thumb" rule that you've read in this series, and elsewhere on the Internet. The rule is to assign 5GB of RAM for every 1TB of disk. This ratio comes from the fact that a deduplicated block seems to occupy about 320 bytes of storage in RAM, and your blocks could occupy anwhere between 512 bytes to 128KB, usually averaging about 64KB in size. So, the ratio sits around 1:208, which is where we come up with the "5GB RAM per 1TB disk" metric. So with a 10TB pool, we can expect to need 50GB of RAM for the DDT, or 200GB of RAM for the ARC.

In both cases, these RAM installations might just be physically or cost prohibitive. In my servers, the motherboards do not allow for more than 32GB of physically installed RAM modules. So 40GB isn't doable. As such, is deduplication out of the question? Not necessarily. If you have a fast SSD, something capable of 100k IOPS, or roughly the equivalent of your RAM install, then you can let the DDT spill out of RAM onto the L2ARC, and performance will not be impacted. A 256GB SSD is much more practical than 200GB of physical RAM modules, both in terms of physical limitations and cost prohibition.

Enterprise Hardware

Without SSD
15k SAS drives don't come cheap. Currently, the Seagate Cheetah drives go for about $1 per 3GB, or about $330 per 1TB. So, for the 8TB we would be spending on disk, we would be spending about $2600 for disk. We already determined that we need about 200GB of space for the ARC. If we need to fit everything in RAM, and our motherboard will support the install size, then ECC registered RAM goes for about $320 per 16GB (how convenient). I'll need at least 14 sticks of 16GB RAM modules. This would put my RAM cost at about $4480. Thus my total bill for storage only would be $7080. I'm only saving $670 by not purchasing 2 disks to save on deduplication.

With SSD
Rather than purchasing 14 16GB memory modules, we could easily purchase an enterprise 256GB fast SLC SSD for about $500. A 256GB SSD is attractive, because as an L2ARC, it will be storing more than just the DDT, but other cached pages from disk. The SSD could also be partitioned to store the ZIL, acting as a SLOG. So, we'll only need maybe 16GB installed RAM (2x8GB modules for dual channel), which would put our RAM cost at

$320, our SSD cost at $500 and our drive cost at $2600, or $3420 for the total setup. This is half of the initial price using only ECC RAM to fit the DDT. That's significant, IMO. Again, I only saved $670 by not purchasing 2 disks.

Commodity Hardware

Without SSD
7200 SATA drives come cheap these days. ZFS was designed with commodity disk in mind, knowing it's full of failures and silent data corruption. I can purchase a single 2TB disk for $80 right now, brand new. Four of those put my total cost at $320. However, the ECC RAM doesn't change, and if I needed 14 of the 16GB sticks as with my enterprise setup, then I can count on my total cost for this commodity setup at $4800. But, a RAM install of that size does not make sense for a "commodity" setup, so let's reduce the RAM footprint, and add an SSD.

With SSD
A fast commodity SSD puts us at the MLC SSDs. The 256GB Samsung 840 Pro is going for $180 right now, and can sustain 100k IOPS, which could possibly rival your DDR3 RAM. So, again sticking with 4 2TB drives at $320, 16GB of RAM at $320 and our Samsung SSD at $180 our total cost for this setup is $820, only saving $80 by not purchasing an additional 2TB SATA drive. This is by far the most cost effective solution.

Additional Hidden Costs & SSD Performance Considerations

When we made these plans on purchasing RAM, we were only considering the cost of storing the ARC and the DDT. We were not considering that your operating system will still need room outside of the ARC to operate. Most ZFS administrators I know won't give more than 25% of RAM to the ARC, on memory intensive setups, and no more than 50% on less memory intensive setups. So, for our 200GB ARC requirement, it may be as much as 400GB of RAM, or even 800GB. I have yet to administer a server with that sort of RAM install. So, SSDs all of the sudden become MUCH more attractive.

If you decide to go the route of an SSD for an L2ARC, you need to make sure that it performs on par with your installed RAM, otherwise you'll see a performance hit when doing lookups in your DDT. It's expected for DDR3 RAM to have a rate of 100k to 150k sustained sequential read/write IOPS. Getting an SSD to perform similarly means getting the high end SATA connected SSDs, or low end PCIe connected, such as the OCZ RevoDrive.

However, suppose you don't purchase an SSD that performs equally with your DDR3 modules. Suppose your DDR3 sustains 100k IOPS, but your SSD only does 20k IOPS. That's 5x as slow as DDR3 (spinning 7200 RPM disk only sustains about 100 IOPS). With as frequently as ZFS will be doing DDT lookups, this is a SIGNIFICANT performance hit. So, it's critical that your L2ARC can match the same bandwidth as your RAM.

Further, there's a hidden cost with SSDs, and that's reliability. Typical enterprise SLC SSDs can endure about 10k write cycles, with wear leveling, before the chips begin to wear down. However, for commodity, more "consumer grade" SSDs, they will only sustain about 3k-5k write cycles. Don't fool yourself though. For our 256GB SSD, this means you can write 256GB 10,000 times, or 2.56PB worth of data on a SLC SSD, or 256GB 3,000 times, or 768TB on an MLC SSD. That's a lot of writing, assuming again, that the SSDs have wear leveling algorithms on board. But, the SSD may fail early, which means the DDT spilling to disk, and completely killing performance of the pool. By putting a portion of the DDT on the SSD, the L2ARC becomes much more write intensive, as ZFS expands the DDT table for new deduplicated blocks. Without deduplication, the L2ARC is less write intensive, and should be very read intensive. So, by not using deduplication, you can lengthen the life of the SSD.

Conclusion

So, now when you approach the CFO with your hardware quote, with a dedup ratio of 1.24:1, it's going to be a hard sell. With commodity hardware using an SSD, you're getting close to your savings in disk (10.25:1), as compared to enterprise hardware where you're spending much, much more to get close to those space savings (5.10:1). But, with commodity hardware, you're spending 1/4 of the enterprise equivalent. In my opinion, it's still too costly.

However, if you can get your ratio close to 2:1, or better, then it may be a good fit. You really need to know your data, and you really need to be able to demonstrate that you will get solid ratios. A storage server of virtual machines might be a good fit, or where you have redundant data that is nearly exactly the same. For general purpose storage, especially with a ratio of 1.24:1, it doesn't seem to be worth it. However, you're not out of luck, if you wish to save disk.

For good space savings on your disk, that you get nearly for free, I strongly encourage compression. Compression doesn't tax the CPU, even for heavy workloads, provides similar space savings (in my example above, I am getting a 1.3:1 compression ratio versus 1.24:1 dedup ratio), doesn't require an expensive DDT, and actually provides enhanced performance. The extra performance comes from the fact that highly compressable data does not need to physically write as much data to slow spinning platter, and also does not need to read as much physical disk. Your spinning disk is the slowest bottleneck in your infrastructure, so anything you can do to optimize the reads and writes could provide large gains. Compression wins here.

Hopefully, this post helps you analyze your deduplication plans, and identify the necessary costs.

Posted by Aaron Toponce on Wednesday, December 18, 2013, at 2:09 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. Gavin | December 18, 2013 at 2:16 am | Permalink

   "zfs -S" in your simulation test should actually be "zdb -S". dedup simulation is a feature of the ZFS debugger rather than the standard ZFS command set.

2. Aaron Toponce | December 18, 2013 at 7:43 am | Permalink

   Yes. Typo. Fixed. Thanks. 🙂

3. Miguel | February 1, 2014 at 12:25 pm | Permalink

   Thank you for the excellent series Aaron.

   Have you considered converting the series into a downloadable PDF? It would make reading offline much easier (on an iPad for instance).

   Thanks

4. John Adams | March 11, 2014 at 12:52 pm | Permalink

   Typo: This can bee a hard puzzle to solve. Bee? 🙂

Thanks for the excellent info.
John

5. defdefred | April 23, 2014 at 5:03 pm | Permalink

   Dedup is mostly efficient for making backup on disk like VTL appliance.
   Monthly or weekly full backups have lots of identical block. Daily incremental backups are also good candidate!

   Dedup can be a way to mimic thin provisioning if all unused data are all zeroed.

6. Mark | March 1, 2015 at 9:19 am | Permalink

   What happens when the SSD used breaks? The L2ARC becomes unavailable, which should be fine. But what about the DDT? Does it get regenerated?

7. Aaron Toponce | March 3, 2015 at 12:03 pm | Permalink

   Yes. The DDT is part of the cache, so on reads and writes, it gets regenerated as needed. All the DDT information resides in the metadata on disk. Keeping the DDT in RAM keeps the ZFS read/write operations snappy.

8. stoertebeker | July 2, 2016 at 7:43 am | Permalink

   I think there is a little mistake in "Determining RAM Needs".
   In the first paragraph you wrote that "that the deduplication table (DDT) will occupy no more than 25% of installed RAM". After the example you wrote "DDT is no more than 25% of ARC". So there is a little difference between 25% of the RAM or the ARC 🙂
   As far as I know, DDT uses not more than 25% of the ARC and the ARC itself can take up to 50% of the installed RAM (at least on my ubuntu-system with zfs-on-linux this value is the default Value).
   This means that on a system with 16 GB RAM the ARC can use up to 8 GB and the DDT up to 2 GB of it.

9. asmo | June 21, 2017 at 2:03 pm | Permalink

   Why is there such a big difference in "Determining RAM Needs" between "Counting Blocks" and "Rule of the Thumb"?

   In the first example the ARC of a 2TB zpool should have an ARC size of 160GB RAM - that's 80GB per terabyte. As well that zpool occupies only 40% of the available space.
   In the second example the 10TB zpool needs 200GB RAM - that's 20GB per terabyte. I guess this calculation based on the assumption that the pool contains ~40% payload data, isn't it?

   So 80GB versus 20GB per terabyte - that's a difference of 4:1.