Becoming a ZFS Ninja

Agenda

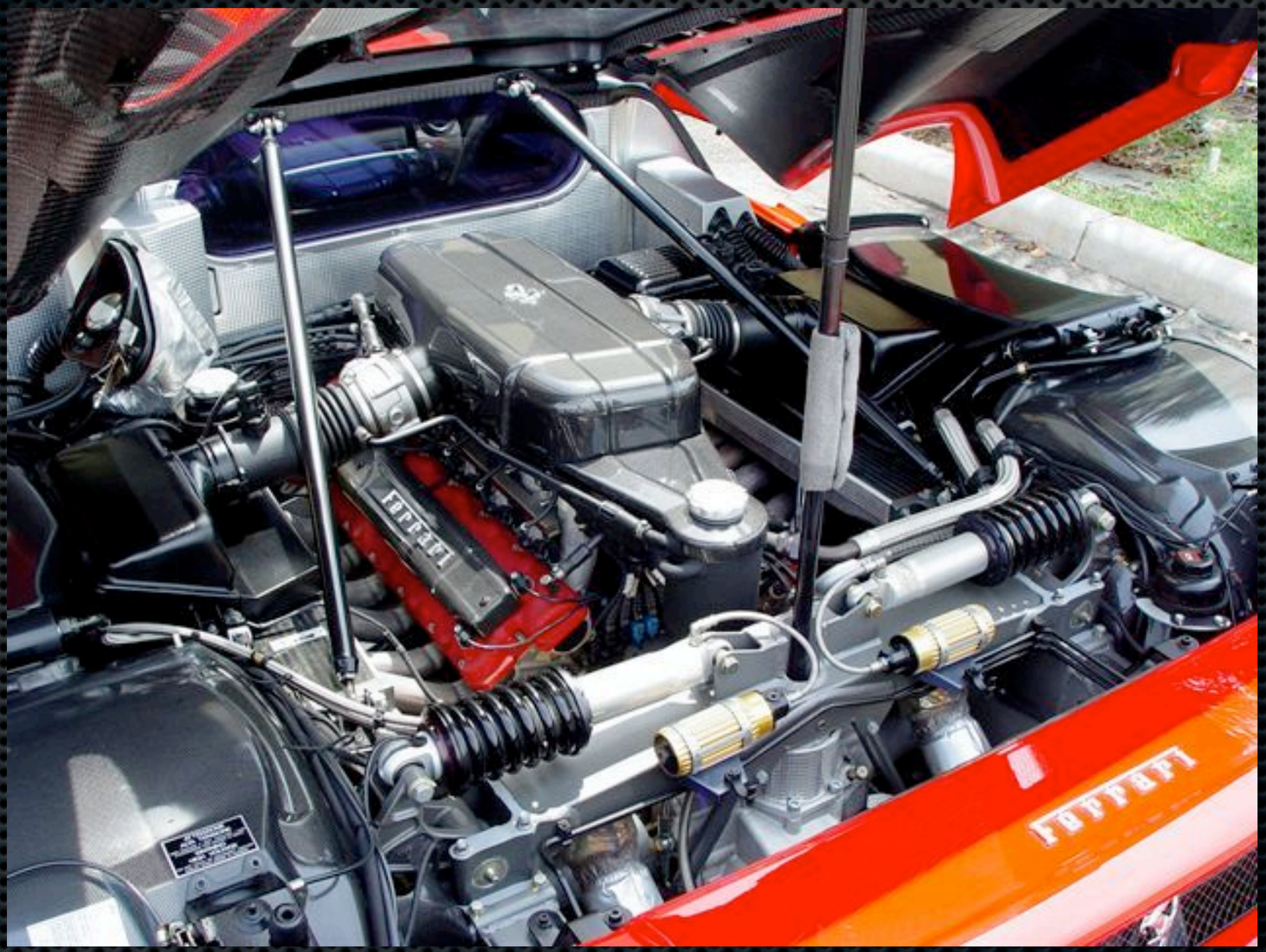
- Attacking the Physical; Pools, RAID, and Maintenance
- Harnessing the Virtual; Datasets and Properties
- Mastering Time & Space; Snapshots, Clones, Replication
- Sharing the Joy; NFS, CIFS, and iSCSI
- Honing the Craft; Internals and Tuning
- Advancing the Fight; NDMP, COMSTAR, etc.

benr@cuddletech.com

Some Basics

- ZFS is a powerful integrated storage sub-system
- ZFS is transactional Copy-on-Write, *always* consistent on disk (no fsck)
- ZFS is scalable, 128bit
- ZFS is fully checksummed
- ZFS is revolutionary and modern, from the ground up
- ZFS loves memory and SSD, and knows how to use them

THINK THIS:



Even though it feels like this:

60 Second Primer

- There are only 2 commands: *zpool* and *zfs*
- Getting started is as easy as....
  - Create a storage “pool” from one or more devices
  - Create additional filesystems and customize them

DEMO:

Creating Simple Pools

Attacking the Physical

ZFS Pools

- ZFS combines the traditional roles of Volume Manager (RAID) and File System
- Born from the idea that disk should work similar to DIMMs, just plug and use.
- Dumps the old 1:1, volume/filesystem, mentatlity
- Controlled with a single command: *zpool*

ZPool Componants

- Pool is created from one or more “vdevs” (Virtual Devices); which is a collection of one or more physical devices.
- Vdev types include:
  - disk: A real disk (c0t0d0)
  - file: A file (/vdevs/vdisk001)
  - mirror: Two or more disks mirrored together
  - raidz1/2: Three or more disks in RAID5/6*
  - spare: A spare drive
  - log: A write log device (ZIL SLOG; typically SSD)
  - cache: A read cache device (L2ARC; typically SSD)

RAID Concepts

- Dynamic Stripe: Intelligent RAID0
- Mirror: n-way RAID1
- RAIDZ1: Improved form of RAID5
- RAIDZ2: Improved form of Dual Parity RAID5 (aka RAID6)
- Combine as dynamic stripe, such as stripe of 2 RAIDZ2

ZPool Create Syntax (Simple)

```
zpool create mypool c0t0d0 c0t1d0 c0t2d0
```

ZPool Create Syntax (Simple)

zpool create mypool c0t0d0 c0t1d0 c0t2d0

Dynamic Stripe: RAID0

ZPool Create Syntax (Moderate)

```
zpool create mypool
    mirror c0t1d0 c1t1d0
        mirror c0t2d0 c1t2d0
```

ZPool Create Syntax (Moderate)

```
zpool create mypool
    mirror c0t1d0 c1t1d0
        mirror c0t2d0 c1t2d0
```

Dynamically Stripped Mirrors: RAID1+0

ZPool Create Syntax (Complex)

```
zpool create MyPool
        mirror c0t0d0 c1t0d0
        mirror c0t1d0 c1t0d0
          raidz c2t1d0 c2t2d0 c2t3d0
            log c4t0d0
          cache c4t1d0
            spare c6t0d0 c6t1d0
```

ZPool Create Syntax (Complex)

zpool create MyPool
mirror c0t0d0 c1t0d0
mirror c0t1d0 c1t0d0
raidz c2t1d0 c2t2d0 c2t3d0
log c4t0d0
cache c4t1d0
spare c6t0d0 c6t1d0

Hybrid Pool of Terror: 1/1/5+0 ??

File VDevs: Play Time

- Files can be used in place of real disks; must be at least 128MB in size, pre-allocated
- Best practice is to store them in a common location
- Not intended for production use!!!

DEMO:

Playing with Pools

Scrubs

- *zpool scrub* should be run periodically
- Verifies on-disk contents and fixes problems if possible
- Schedule via cron to run weekly, on off hours

Pool Import/Export

- Typically done during system boot/halt.
- Pools can be moved from one play to another

ZPool Maintenance

- Manually spare pool disks with:
  - *zpool replace <old_dev> <new_dev>*
- Devices can be online'd offlined, via *zpool online <dev>* or *zpool offline <dev>*
- A device rebuild is known as a "resilver"
- *zpool attach* and *zpool detach* provided to add or remove devices from a redundant group (mirror)

Growing ZPools

- Add additional capacity using *zpool add <vdev>*
- Syntax is similar to create:
  - *zpool add mypool mirror c6t0d0 c7t0d0*
  - *zpool add mypool spare c5t0d0 c5t1d0*
  - *zpool add mypool log c4t0d0*
  - *zpool add mypool raidz2 c2t1d0 c2t2d0 c2t3d0...*
- You can not shrink a pool (today)

ZPool Properties

- Each pool has customizable properties
- Important properties are: failmode, autoreplace, and listsnapshots

```
root@quadra vdev$ zpool get all quadra
NAME    PROPERTY       VALUE      SOURCE
quadra  size           1.09T      -
quadra  used           634G       -
quadra  available      478G       -
quadra  capacity       57%        -
quadra  altroot        -          default
quadra  health         ONLINE     -
quadra  guid           11193326520898087025  default
quadra  version        14         default
quadra  bootfs         -          default
quadra  delegation     off        local
quadra  autoreplace    off        default
quadra  cachefile      -          default
quadra  failmode       wait       default
quadra  listsnapshots  on         local
```

ZPool History

- zpool history command provides excellent history to find out what happened when.

```
$ zpool history quadra
History for 'quadra':
2008-10-09.11:29:36 zpool create quadra raidz c0d1 c2d0 c3d0
2008-10-09.11:31:47 zfs recv -d quadra
2008-10-09.11:32:20 zfs create quadra/home
2008-10-09.11:32:37 zfs set atime=off quadra
2008-10-09.11:32:46 zfs set compression=on quadra/home
2008-10-09.13:44:33 zfs recv -d quadra/home
```

ZPool Sizing

- ZFS Reserves 1/64th of pool capacity for safe-guard to protect COW
- RAIDZ1 Space is Drive's Total Capacity - 1 Drive
- RAIDZ2 Space is Drive's Total Capacity - 2 Drives
- So...
  - Dyn. Stripe of 4* 100GB= 400 / 1.016= ~390GB
  - RAIDZ1 of 4* 100GB = 300GB - 1/64th= ~295GB
  - RAIDZ2 of 4* 100GB = 200GB - 1/64th= ~195GB
  - RAIDZ2 of 10* 100GB = 800GB - 1/64th= ~780GB

Harnessing the Virtual

ZFS Datasets

- A *dataset* is a control point, that come in two forms:
  - filesystem: Traditional POSIX filesystem
  - volume: Raw data store
- Intended to be nested
- Each dataset has associated properties which can be inherited by sub-filesystems (parent-child relationship)
- Controlled with *zfs* command

Filesystem Datasets

- Create a new dataset with *zfs create mypool/dataset*
- Datasets which act as holder for other datasets are called “stubs”
- New datasets inherit the properties of the parent dataset, where appropriate; such as mountpoint, etc.

Dataset Properties

- Get all properties with *zfs get all pool/ds*
- Set a given properties with *zfs set key=value pool/ds*
- Properties come in 3 varieties:
  - Native Read Only: Informational, such as “used”
  - Native Read/Write: Configuration, such as “quota”
  - User Properties: Custom created, such as “Description”

Volume Datasets (ZVols)

- Block storage; devices are */dev/zvol/rdsk/pool/ds*
- Can be used for iSCSI or creating non-ZFS local filesystems such as UFS, ext2, xfs, etc.
- Can be created “sparse”, known as “thin provisioning”; disk isn’t allocated until its required.
- Shares many of the same properties as filesystem datasets

DEMO:

Fun with Datasets

Mastering of Time & Space

Snapshots

- Snapshots are a natural benefit of ZFS's Copy-On-Write design
- Creates a point-in-time "copy" of a dataset
- Can be used for individual file recovery or full dataset "rollback"
- Fundamental building block of advanced functionality
- Denoted by @ symbol
- Created like so: *zfs snapshot pool/my/dataset@now*
- Snapshots consume no additional disk space*

Hidden Goodness

- Each dataset mountpoint has a hidden “.zfs” directory
  - Can be exposed by changing “snapdir” property from “hidden” to “visible”
- Within it a snapshots/ directory which contains the snapshot names
- Can navigate snapshots for data you want and copy out in the normal way (Read-Only of course)

Rollback

- Irreversibly revert a dataset to a previous snapshot state
- Example: To recover from a hack on your database
- All snapshots more current than the one to which you rollback are lost

Clones

- Clone turns a snapshot into a separate Read-Write dataset
- The clone is acts like any normal dataset, but consumes no disk space
- Can be used to:
  - Access old data for testing without disturbing the real data
  - Quickly replicate an application environment; ie: Virtualization (xVM, Zones, VirtualBox, etc.)
- Caveat! Is still dependent on “origin” snapshot!

Promoting Clones

- Cloned datasets maintain a relationship with its parent snapshot
- You can sever this relationship by promoting it to its own autonomous state
- Clone is no longer a clone, its an autonomous dataset

Putting it together: Example

- Via cron, you snapshot your database hourly
- During the night, the database is hacked and damaged
- You want to get back to business but investigate more:
  - Clone a snapshot of the hacked database
  - Promote it
  - Revert the origonal dataset to a snapshot prior to the hack

DEMO:

Snapshots & Friends

Replication

- Dataset Snapshots can be **sent** and **recieved**
- *zfs send* produces a data stream of the snapshot which can be piped to *zfs recv*
- Example:

```
# zfs send pool/ds@123 | ssh root@sys zfs recv pool/dsbackup
```

Sharing the Joy

Sharing Made Easy

- ZFS integrates with NFS, CIFS, and iSCSI
- Share a dataset by simply turning a property “on”, ZFS does all the heavy lifting for you

Sharing via NFS

- Controlled by “sharenfs” dataset property
- Property is inherited by child datasets
- Simple form: *zfs set sharenfs=on pool/mydataset*
- Advanced form: Pass “export” -o arguments as value (*see share_nfs(1M)*)

zfs set sharenfs=”ro=192.168.100/24,...” pool/mydataset

Sharing via iSCSI

- Can only share Volume datasets (obviously)
- Controlled by “shareiscsi” property
- Can be turned “on” or “off”
- Example: *zfs set shareiscsi=on pool/MailVolume*
- Currently uses “old” iSCSI Target, see *iscsitadm(1M)*
  - *ie: Not COMSTAR*

Sharing via SMB (CIFS)

- Controlled by “sharesmb” property
- Example: *zfs set sharesmb=on pool/WinHomes*
- Relies on Samba
- Related dataset properties:
  - *nbmand* (Non-Blocking mandatory locks)
  - *vscan*
  - *casesensitivity*

DEMO:

Sharing NFS & iSCSI

Honing the Craft

Record Size

- ZFS uses a variable block size
- Blocks larger than the record size are broken up
- Default record size is 128K
- Therefore....
  - 1K write consumes one 1K block
  - 64K write consumes one 64K block
  - 265K write consumes two 128K blocks
- Change to 8K record size for databases

ARC

- Adaptive Replacement Cache; the ZFS Read Cache
- Exists in physical memory
- Can dynamically grow and shrink
- Very intelligent, maintains 4 cache lists: Most Recently Used (MRU) and Most Frequently Used (MFU)
- Can consume up to 7/8th of physical memory!
- Tunable!

ARC Kstats

- Several Kstats provide a window into the ARC
- Use ‘arc_summary’ to simplify interpretation
  - http://cuddletech.com/arc_summary/

Prefetch

- Intelligently fetch blocks before you ask for them
- The more you fetch the more it prefetches (up to 256 blocks)
- Feeds ARC
- Can be disabled; but don’t.

VDev Read-Ahead

- Meta-data disk reads less than 16K are expanded to 64K and stored in a per-vdev 10MB LRU cache
- Low overhead, improves disk performance by leveraging memory
- Tunable, but doesn't require tuning.

Cache Flush

- ZFS can manage caches on storage devices by ordering them to flush after critical operations
- Can be turned off, but should only be done for very large self-managing battery backed storage devices (ie: 1GB EMC Symmetrix LUN)
- Gotten more intelligent over time, avoid tuning.

Transaction Groups (TXG)

- I/O requests are treated as “transaction”, similar to a transactional database
- Transactions are commited to a “Transaction Group”
- There are 3 groups at all times: Open, Quiescing, and Syncing.
- TXG Syncing occurs every 30 seconds (tunable)

Monitoring I/O

- Two ways to monitor:
  - Virtual File System (VFS) Layer; use *fsstat*
  - Physical Disk I/O; use *iostat*
- VFS Layer Data is whats important, NOT Physical I/O!

Write Throttle

- Processes that are writing data too quickly will be throttled by 1 tick each request to slow ingress of data
- Impacts only the culprate
- Is tunable, can be disabled.

ZFS Intent Log (ZIL)

- TXG Sync delay is only appropriate for asynchronous writes
- Synchronous writes must be commited to “stable storage” before control is returned to caller. (blocking)
- ZFS will immediately “log” such writes to more quickly return control.
- Can be disabled, but dangerous.
- Usually written to zpool, but can be directed to a dedicated high-speed disk (SSD)

Hybrid Pools

- A hybrid pool is one which utilizes SSDs as a middle ground between super-fast DRAM and slow disk.
- Used in two ways:
  - L2ARC: Level-2 ARC, to expand your read cache
  - ZIL: Dedicated ZIL disk(s); aka “SLOGs”

Creating a Hybrid Pool

- Add L2ARC Device
  - *zpool add mypool cache c9t0d0*
  - *zpool add mypool cache mirror c9t0d0 c9t1d0*
- Add ZIL SLOG
  - *zpool add mypool log c9t1d0*
  - *zpool add mypool log mirror c9t1d0 c9t2d0*

mdb

- Many ZFS tunables can be changed dynamically via mdb
- Can be used to view all tunables, ala “::zfs_params”

```
> ::zfs_params
arc_reduce_dnlc_percent = 0x3
zfs_arc_max = 0x0
zfs_arc_min = 0x0
arc_shrink_shift = 0x5
zfs_mdcomp_disable = 0x0
zfs_prefetch_disable = 0x0
```

More Tuning Info

- Blogs (Google is your friend)
- Mailing Lists (opensolaris.org)
- Solaris Internals Wiki: ZFS Evil Tuning Guide

Advancing the Fight

NDMP

- Solaris NDMP is “zfs aware”
- If you backup a ZFS dataset via NDMP, a snapshot will be created for the duration of the backup
- See *ndmpadm(1M)* for information on enabling NDMP

COMSTAR

- Common Multiprotocol SCSI Target
- Essentially a storage router;
  - SCSI Logical Units are created from backing storage such as a ZVol
  - Port Providers emulate a protocal Target, such as Fibre Channel, SAS, iSCSI, etc.
  - The *SCSI target mode framework* provides a controlled bridge between the two.
- Meaning... any Solaris box can look like a Fibre Channel Array

Thank You.

Ben Rockwood
Joyent, Inc

web: cuddletech.com
email: benr@cuddletech.com

Robert Milkowski
Jason Williams
Marcelo Leal
Max Bruning
James Dickens
Adrian Cockroft
Jim Mauro
Richard McDougall
Tom Haynes
Peter Tribble
Jason King
Octave Orgeron
Brendan Gregg
Matty
Roch
Joerg Moellenkamp
...