

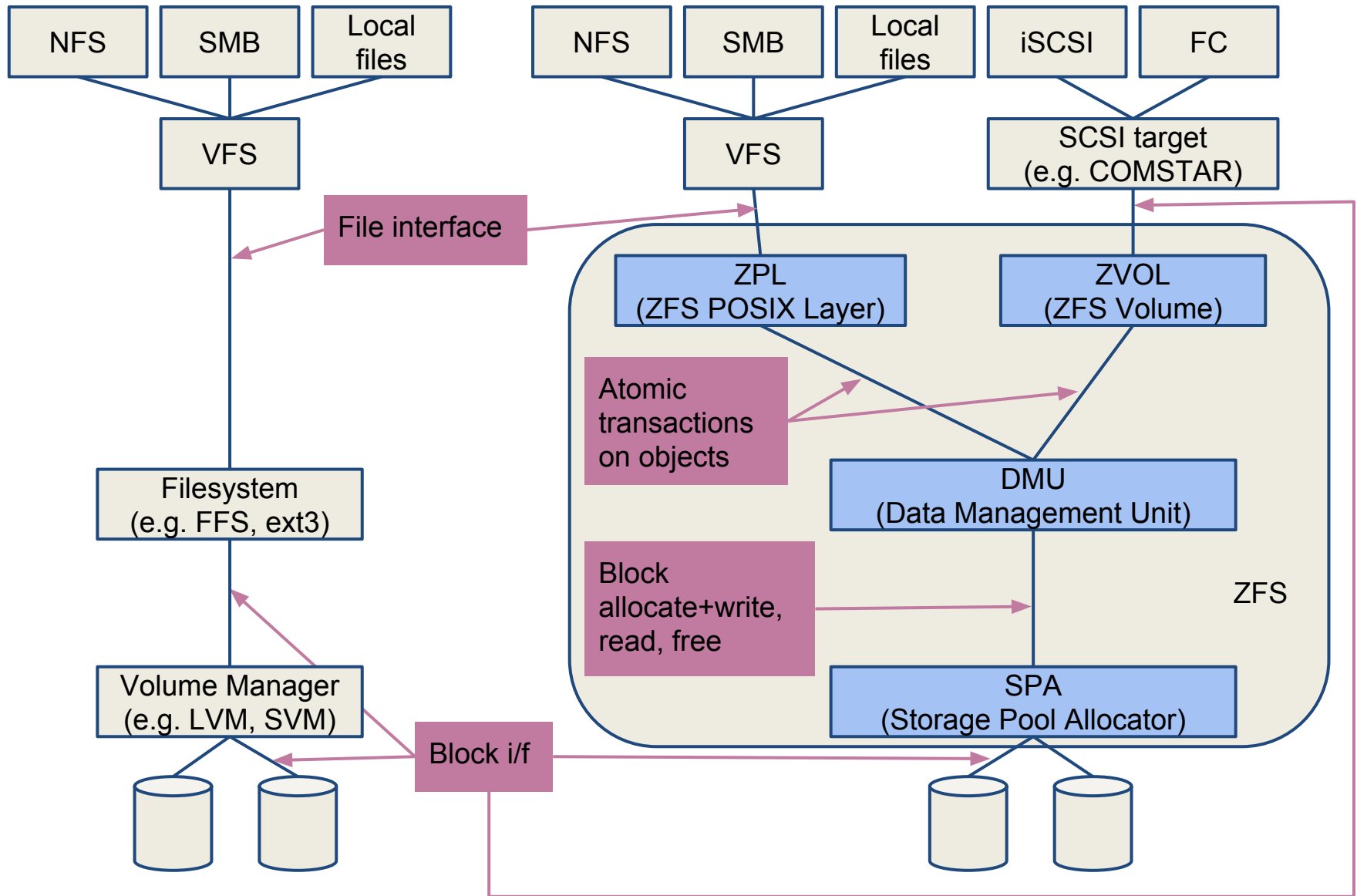
OpenZFS

Matt Ahrens
mahrens@delphix.com



What is the ZFS storage system?

- Pooled storage
 - Functionality of filesystem + volume manager in one
 - Filesystems allocate and free space from pool
- Transactional object model
 - Always consistent on disk (no FSCK, ever)
 - Universal - file, block, NFS, SMB, iSCSI, FC, ...
- End-to-end data integrity
 - Detect & correct silent data corruption
- Simple administration
 - Concisely express intent
 - Scalable data structures



ZFS History

- 2001: development starts with 2 engineers
- 2005: ZFS source code released
- 2008: ZFS released in FreeBSD 7.0
- 2010: Oracle stops contributing source code for ZFS
- 2010: illumos is founded
 - The multilateral successor to OpenSolaris
- 2013: ZFS on (native) Linux GA
- 2013: Open-source ZFS bands together to form OpenZFS
- 2014: OpenZFS for Mac OS X launched

What is OpenZFS?

OpenZFS is a community project founded by open source ZFS developers from multiple operating systems:

- illumos, FreeBSD, Linux, OS X, OSv

The goals of the OpenZFS project are:

- to **raise awareness** of the quality, utility, and availability of open source implementations of ZFS
- to encourage **open communication** about ongoing efforts to improve open source ZFS
- to ensure **consistent** reliability, functionality, and performance of all distributions of ZFS.

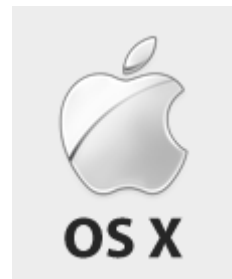
OpenZFS activities

<http://open-zfs.org>

- Platform-independent [mailing list](#)
 - Developers discuss and review platform-independent code and architecture changes
 - Not a replacement for platform-specific mailing lists
- Simplifying the [illumos development process](#)
- Creating cross-platform test suites
- Reducing [code differences](#) between platforms
- [Office Hours](#) a.k.a Ask the Expert



OpenZFS



Last 12 months features

- embedded data blocks (better compression)
- larger (1MB+) blocks
- better ENOSPC error handling
- metadata_redundancy=most
- LZ4 by default (for metadata, compress=on)
- metaslab fragmentation metric
- UNMAP perf improvements
- L2ARC memory overhead halved
- ARC lock contention
 - 3x improvement in cached reads
- ARC_no_grow fixes

Upcoming features

Implemented, pending integration:

- prefetch rewrite
- compressed ARC
- resumable zfs send/recv
- allocation throttle
- recv prefetch
- new checksum algos (SHA512, Skein, Edon-R)

Work in progress:

- device removal
- persistent l2arc (Saso Kiselkov)
- channel program for richer administration

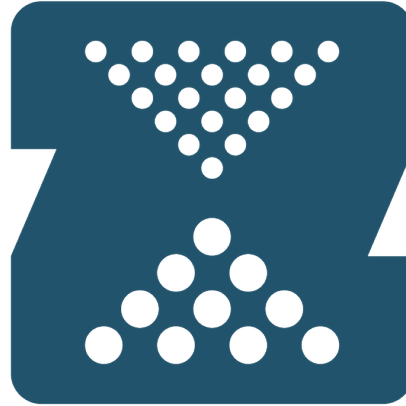
Last 12 months events

- May 2014
 - OpenZFS European Conference, Paris
 - dotScale, Paris
 - BSDCAN, Ottawa, Canada
- November 2014
 - OpenZFS Developer Summit, San Francisco
- March 2015
 - AsiaBSDcon, Tokyo
 - Snow UNIX Event, Netherlands

Announcing 2015 OpenZFS Developer Summit

- October 19th-20th
- Downtown San Francisco
- One day of talks
- One day hackathon
- Talk proposals due August 31st
- A few sponsorship opportunities remain
- New this year: \$50 registration fee
 - Registration will open in a few months





OpenZFS

<http://open-zfs.org>

Matt Ahrens
mahrens@delphix.com

ZFS send / receive

- Use cases
- Compared with other tools
- How it works: design principles
- New features since 2010
 - send size estimation & progress reporting
 - holey receive performance!
 - bookmarks
- Upcoming features
 - resumable send/receive
 - receive prefetch

Use cases - what is this for?

- “zfs send”
 - serializes the contents of snapshots
 - creates a “send stream” on stdout
- “zfs receive”
 - recreates a snapshot from its serialized form
- Incrementals between snapshots
- Remote Replication
 - Disaster recovery / Failover
 - Data distribution
- Backup

Examples

```
zfs send pool/fs@monday | \
ssh host \
zfs receive tank/recvd/fs


zfs send -i @monday \
pool/fs@tuesday | ssh ...
```


Examples

```
zfs send pool/fs@monday | \
ssh host \
zfs receive tank/recvd/fs
```



```
zfs send -i @monday \
pool/fs@tuesday | ssh ...
```

 "FromSnap"

 "ToSnap"

Compared with other tools

- Performance
 - Incremental changes located and transmitted efficiently
 - Including individual blocks of record-structured objects
 - e.g. ZVOLs, VMDKs, database files
 - Uses full IOPS and bandwidth of storage
 - Uses full bandwidth of network
 - Latency of storage or network has **no impact**
- Shared blocks (snapshots & clones)
- Completeness
 - Preserves all ZPL state
 - No special-purpose code
 - e.g. owners (even SIDs)
 - e.g. permissions (even NFSv4 ACLs)

How it works: Design Principles (overview)

- Locate changed blocks via block birth time
 - Read minimum number of blocks
- Prefetching issues of i/o in parallel
 - Uses full bandwidth & IOPS of all disks
- Unidirectional
 - Insensitive to network latency
- DMU consumer
 - Insensitive to ZPL complexity

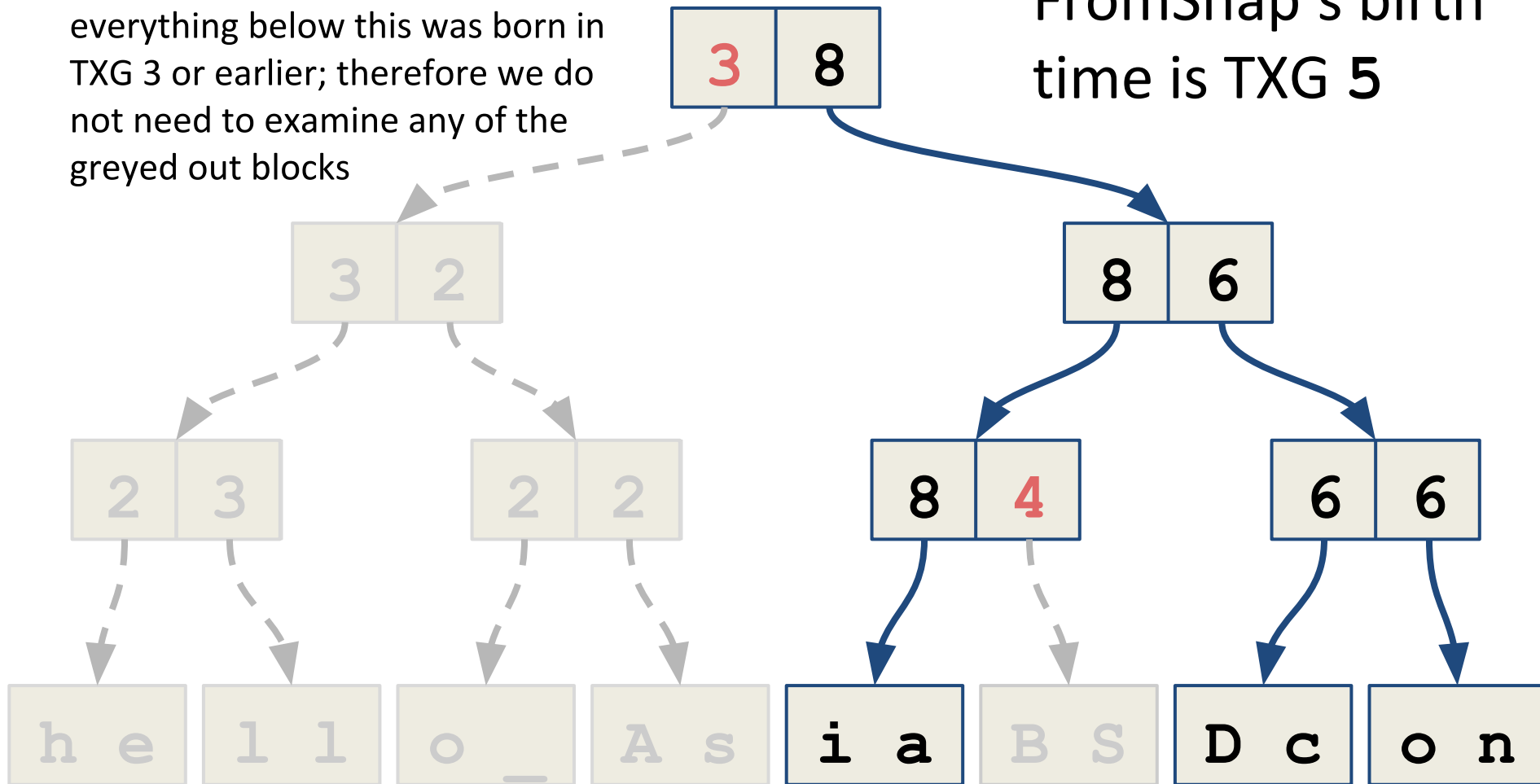
Design: locating incremental changes

- Locate incremental changes by traversing ToSnap
 - skip blocks that have not been modified since FromSnap
- Other utilities (e.g. rsync) take time proportional to # files (+ # blocks for record-structured files)
 - regardless of how many files/blocks were modified
- Traverse ToSnap
- Ignore blocks not modified since FromSnap
- Note: data in FromSnap is not accessed
 - Only need to know FromSnap's creation time (TXG)

Design: locating incremental changes

Block Pointer tells us that everything below this was born in TXG 3 or earlier; therefore we do not need to examine any of the greyed out blocks

FromSnap's birth time is TXG 5



Design: prefetching

- For good performance, need to use all disks
 - Issue many concurrent i/os
- “zfs send” creates prefetch thread
 - reads the blocks that the main thread will need
 - does not wait for data blocks to be read in (just issues prefetch)
 - see tunable `zfs_pd_bytes_max` (default: 50MB)
- Note: separate from “predictive prefetch”

Design: unidirectional

- “zfs send ... | ” emits data stream on stdout
- no information is passed from receiver to sender
- CLI parameters to “zfs send” reflect receiver state
 - e.g. most recent common snapshot
 - e.g. features that the receiving system supports (embedded, large blocks)
- **insensitive to network latency**
- allows use for backups (stream to tape/disk)
- allows flexible distribution (chained)

Design: DMU consumer

- Sends contents of objects
- Does not interpret ZPL / zvol state
- All esoteric ZPL features preserved
 - SID (Windows) users
 - Full NFSv4 ACLs
 - Sparse files
 - Extended Attributes

Design: DMU consumer

```
# zfs send -i @old pool/filesystem@snapshot | zstreamdump
BEGIN record
    hdrtype = 1 (single send stream, not send -R)
    features = 30004 (EMBED_DATA | LZ4 | SA_SPILL)
    magic = 2f5bacbac ("ZFS backup backup")
    creation_time = 542c4442 (Oct 26, 2013)
    type = 2 (ZPL filesystem)
    flags = 0x0
    toguid = f99d84d71cffe4
    fromguid = 96690713123bfc0b
    toname = pool/filesystem@snapshot
END checksum = b76ecb7ee4fc215/717211a93d5938dc/80972bf5a64ad549/a8ce559c24ff00a1
```

SUMMARY:

```
Total DRR_BEGIN records = 1
Total DRR_END records = 1
Total DRR_OBJECT records = 22
Total DRR_FREEOBJECTS records = 20
Total DRR_WRITE records = 22691
Total DRR_WRITE_EMBEDDED records = 0
Total DRR_FREE records = 114
```

Design Principles (DMU consumer)

```
# zfs send -i @old pool/filesystem@snapshot | zstreamdump -v
BEGIN record
. . .
OBJECT object = 7 type = 20 bonustype = 44 blksize = 512 bonuslen = 168
FREE object = 7 offset = 512 length = -1
FREEOBJECTS firstobj = 8 numobjs = 3
OBJECT object = 11 type = 20 bonustype = 44 blksize = 1536 bonuslen = 168
FREE object = 11 offset = 1536 length = -1
OBJECT object = 12 type = 19 bonustype = 44 blksize = 8192 bonuslen = 168
FREE object = 12 offset = 32212254720 length = -1
WRITE object = 12 type = 19 (plain file) offset = 1179648 length = 8192
WRITE object = 12 type = 19 (plain file) offset = 2228224 length = 8192
WRITE object = 12 type = 19 (plain file) offset = 26083328 length = 8192
. . .
```

Send/receive features unique to OpenZFS

- ZFS send stream size estimation
- ZFS send progress monitoring
- Holey receive performance!
- Bookmarks

ZFS send stream size & progress

- In OpenZFS since Nov 2011 & May 2012

```
# zfs send -vei @old pool/fs@new | ...
```

```
send from @old to pool/fs@new estimated size is 2.78G
```

```
total estimated size is 2.78G
```

TIME	SENT	SNAPSHOT
06:57:10	367M	pool/fs@new
06:57:11	785M	pool/fs@new
06:57:12	1.08G	pool/fs@new
...		

- **-P** (parseable) option also available
- API (libzfs & libzfs_core) also available

Holey receive performance!

- In OpenZFS since end of 2013
- Massive improvement in performance of receiving objects with “holes”
 - i.e. “sparse” objects
 - e.g. ZVOLs, VMDK files
- Record birth time for holes
 - Don’t need to process old holes on every incremental
 - `zpool set feature@hole_birth=enabled pool`
- Improve time to punch a hole (for zfs recv)
 - from $O(N \text{ cached blocks})$ to $O(1)$

Bookmarks

- In OpenZFS since December 2013
- Incremental send only looks at FromSnap's creation time (TXG), not its data
- Bookmark remembers its birth time, not its data
- Allows FromSnap to be deleted, use FromBookmark instead

Upcoming features in OpenZFS

- Resumable send/receive
- Checksum in every record
- Receive prefetching
- Open-sourced March 16
 - <https://github.com/delphix/delphix-os>
- Will be upstreamed to illumos

Resumable send/receive: the problem

- Failed receive must be restarted from beginning
 - Causes: network outage, sender reboot, receiver reboot
- Result: progress lost
 - partially received state destroyed
 - must restart send | receive from beginning
- Real customer problem:
 - Takes 10 days to send|recv
 - Network outage almost every week
 - :-(

Resumable send/receive: the solution

- When receive fails, keep state
 - Do not delete partially received dataset
 - Store on disk: last received <object, offset>
- Sender can resume from where it left off
 - Seek directly to specified <object, offset>
 - No need to revisit already-sent blocks

Resumable send/receive: how to use

- Still unidirectional
- Failed receive sets new property on fs
 - **receive_resume_token**
 - Opaque; encodes <object, offset>
- Sysadmin or application passes token to “zfs send”

Resumable send/receive: how to use

- `zfs send ... | zfs receive -s ... pool/fs`
 - New **-s** flag indicates to Save State on failure
- `zfs get receive_resume_token pool/fs`
- `zfs send -t <token> | zfs receive ...`
 - Token tells send:
 - what snapshot to send, incremental FromSnap
 - where to resume from (object, offset)
 - enabled features (embedded, large blocks)
- `zfs receive -A pool/fs`
 - Abort resumable receive state
 - Discards partially received data to free space
 - `receive_resume_token` property is removed
- Equivalent API calls in `libzfs / libzfs_core`

Resumable send/receive: how to use

```
# zfs send -v -t 1-e604ea4bf-e0-789c63a2...
```

```
resume token contents:
```

```
nvlist version: 0
```

```
    fromguid = 0xc29ab1e6d5bcf52f
```

```
    object = 0x856 (2134)
```

```
    offset = 0xa0000 (655360)
```

```
    bytes = 0x3f4f3c0
```

```
    toguid = 0x5262dac9d2e0414a
```

```
    toname = test/fs@b
```

```
send from test/fs@a to test/fs@b estimated  
size is 11.6M
```

Resumable send/receive: how to use

```
# zfs send -v -t 1-e60a... | zstreamdump -v
BEGIN record
...
    toguid = 5262dac9d2e0414a
    fromguid = c29ab1e6d5bcf52f
nvlist version: 0
    resume_object = 0x856 (2134)
    resume_offset = 0xa0000 (655360)
OBJECT object = 2134 type = 19 bonustype = 44 blksize = 128K bonuslen = 168
FREE object = 2134 offset = 1048576 length = -1
...
WRITE object = 2134 type = 19 (plain file) offset = 655360 length = 131072
WRITE object = 2134 type = 19 (plain file) offset = 786432 length = 131072
...
```


Checksum in every record

- Old: checksum at end of stream
- New: checksum in every record ($\leq 128\text{KB}$)
- Use case: reliable resumable receive
 - Incomplete receive is still checksummed
- Checksum verified before acting on metadata

```
# zfs send ... | zstreamdump -vv
```

```
FREEOBJECTS firstobj = 19 numobjs = 13
```

```
    checksum = 8aa87384fb/32451020199c5/bff196ad76a8...
```

```
WRITE_EMBEDDED object = 3 offset = 0 length = 512
```

```
    comp = 3 etype = 0 lsize = 512 psize = 65
```

```
    checksum = 8d8e106aca/34f610a4b5012/cfacccd01ac3...
```

```
WRITE object = 11 type = 20 offset = 0 length = 1536
```

```
    checksum = 975f44686d/3872578352b3c/e4303914087d...
```

Receive prefetch

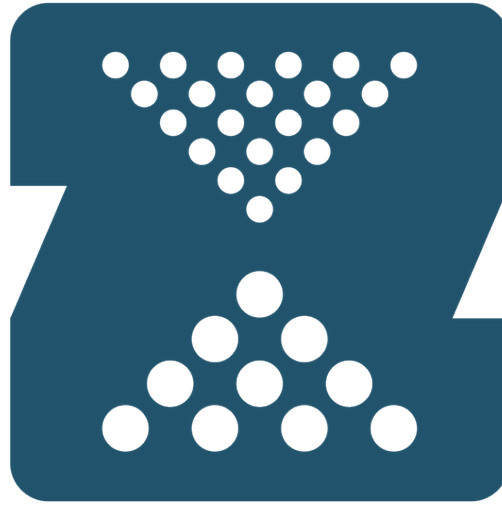
- Improves performance of “zfs receive”
 - Incremental changes of record-structured data
 - e.g. databases, zvols, VMDKs
- Write requires read of indirect that points to it
- Problem: this read happens synchronously
 - get record from network
 - issue read i/o for indirect
 - wait for read of indirect to complete
 - perform write (i.e. notify DMU - no i/o)
 - repeat

Receive prefetch: Solution

- Main thread:
 - Get record from network
 - issue read i/o for indirect (prefetch)
 - enqueue record (save in memory)
 - repeat
- New worker thread:
 - dequeue record
 - wait for read of indirect block to complete
 - perform write (i.e. notify DMU - no i/o)
 - repeat
- Benchmark: **6x faster**
- Customer database: 2x faster

ZFS send / receive

- Use cases
- Compared with other tools
- How it works: design principles
- New features since 2010
 - send size estimation & progress reporting
 - holey receive performance!
 - bookmarks
- Upcoming features
 - resumable send/receive (incl. per-record cksum)
 - receive prefetch



OpenZFS

Matt Ahrens
mahrens@delphix.com

