ZFS Features & Concepts TOI

Andreas Dilger
Lustre Technical Lead

ZFS Design Principles

Pooled storage

- Completely eliminates the antique notion of volumes
- Does for storage what VM did for memory

End-to-end data integrity

- Historically considered “too expensive”
- Turns out, no it isn't
- And the alternative is unacceptable

Transactional operation

- Keeps things always consistent on disk
- Removes almost all constraints on I/O order
- Allows us to get huge performance wins

FS/Volume Model vs. ZFS

Traditional Volumes

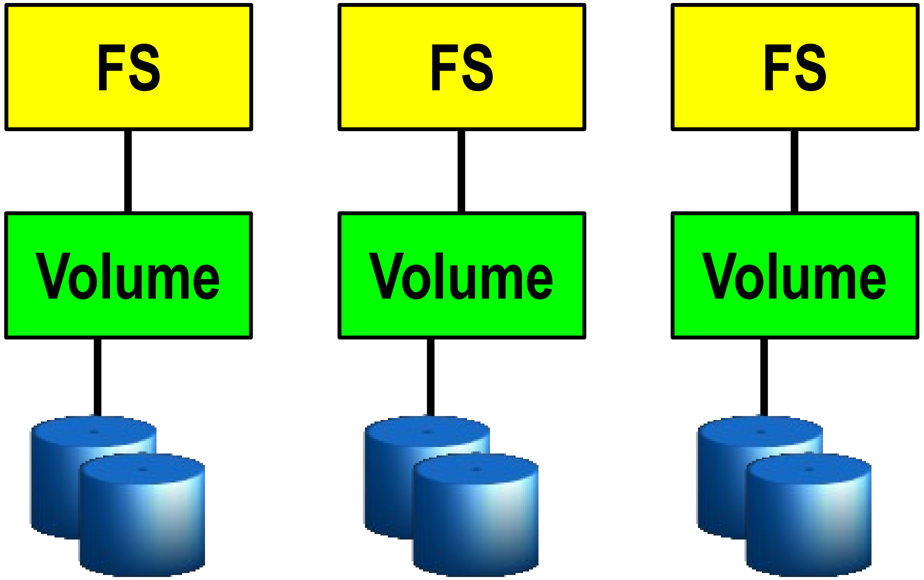
Abstraction: virtual disk

Partition/volume for each FS

Grow/shrink by hand

Each FS has limited bandwidth

Storage is fragmented, stranded



FS FS FS

Volume Volume Volume

ZFS Pooled Storage

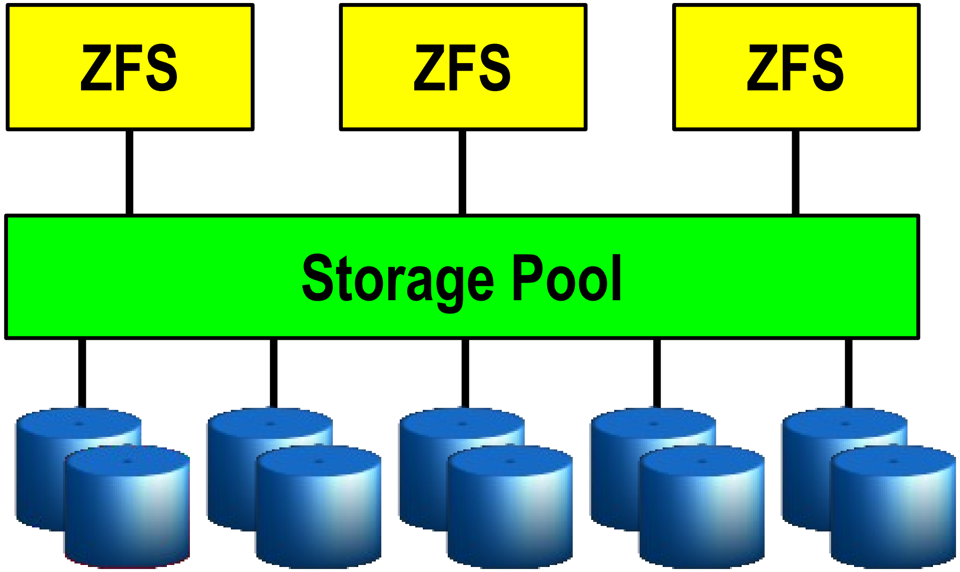
Abstraction: malloc/free

No partitions to manage

Grow/shrink automatically

All bandwidth always available

All storage in the pool is shared

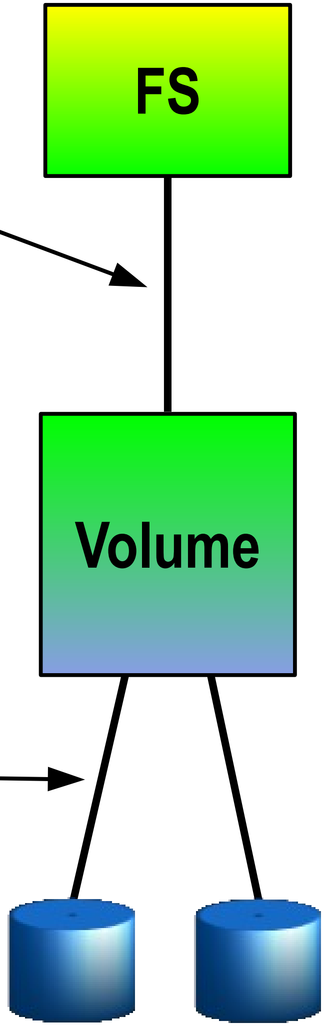


ZFS ZFS ZFS

Storage Pool

FS/Volume Model vs. ZFS

FS/Volume I/O Stack



Block Device Interface

"Write this block, then that block, ..."

Loss of power = loss of on-disk consistency

Workaround: journaling, which is slow & complex

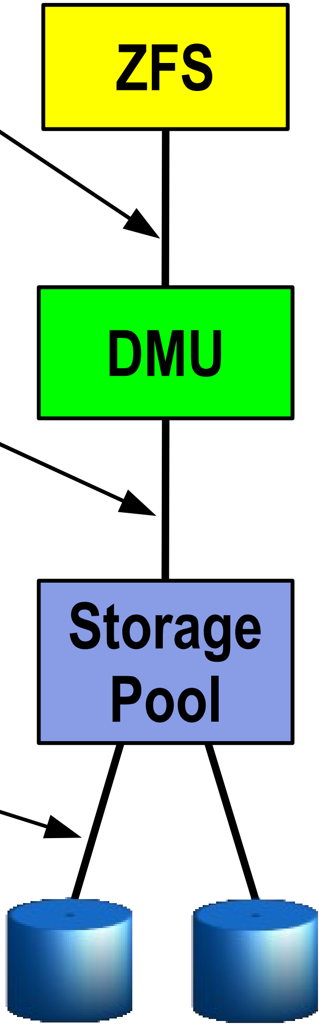
Block Device Interface

Write each block to each disk immediately to keep mirrors in sync

Loss of power = resync

Synchronous and slow

ZFS I/O Stack



Object-Based Transactions

"Make these 7 changes to these 3 objects"

All-or-nothing

Transaction Group Commit

Again, all-or-nothing

Always consistent on disk

No journal – not needed

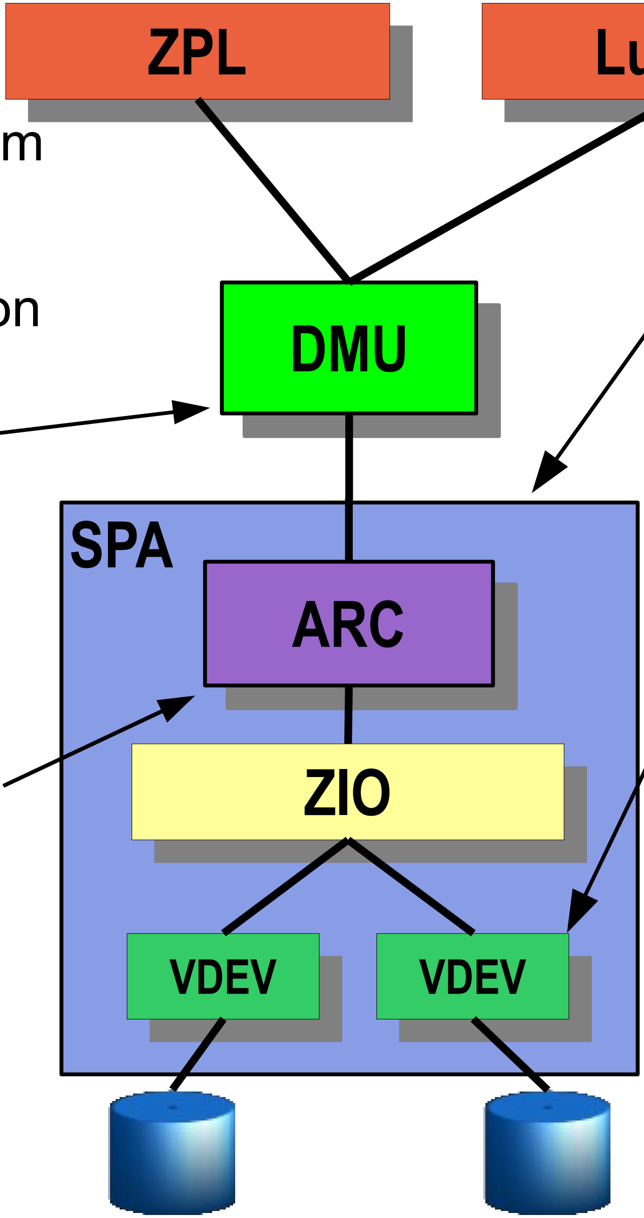
Transaction Group Batch I/O

Schedule, aggregate, and issue I/O at will

No resync if power lost

Runs at platter speed

ZFS IO Stack



POSIX Interface

Look and feel of a file system but much, much, more

VNODE/VFS Implementation

Object-Based Transactions

“Make these 7 changes to these 3 objects”

All-or-nothing

Transaction Group Batch I/O

Schedule, aggregate, and issue I/O at will

No resync if power lost

Runs at platter speed

Transaction Group Commit

Again, all-or-nothing

Always consistent on disk

No journal – not needed

Virtual Devices

Take the block interface to the lowest level

Provide mirroring, raid-z, and spare operations

Dirty Time Logging for quick resilvering

ZFS Data Integrity Model

Everything is copy-on-write

- Never overwrite live data
- On-disk state always valid – no “windows of vulnerability”
- No need for fsck(1M)

Everything is transactional

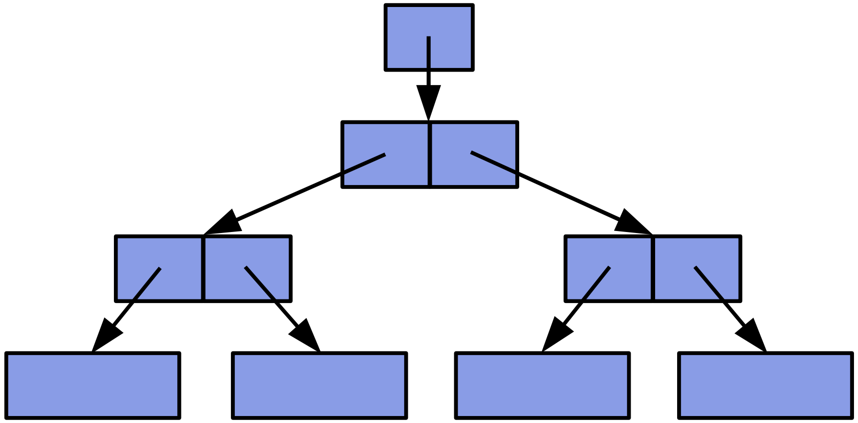
- Related changes succeed or fail as a whole
- No need for journaling

Everything is checksummed

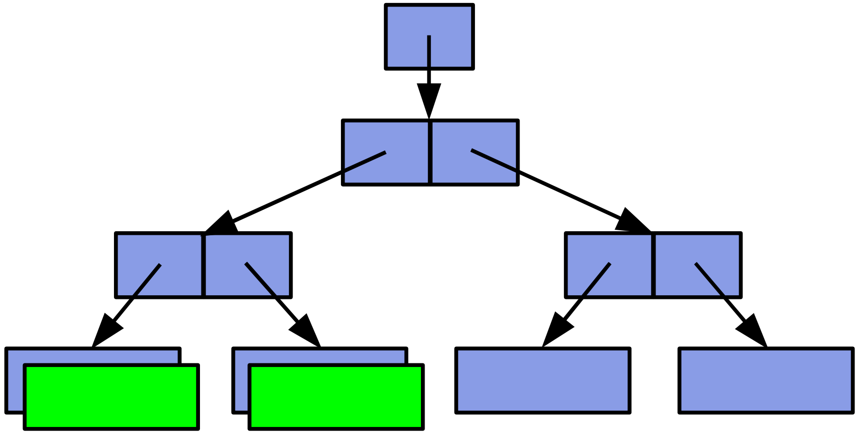
- No silent data corruption
- No panics due to silently corrupted metadata

Copy-On-Write Transactions

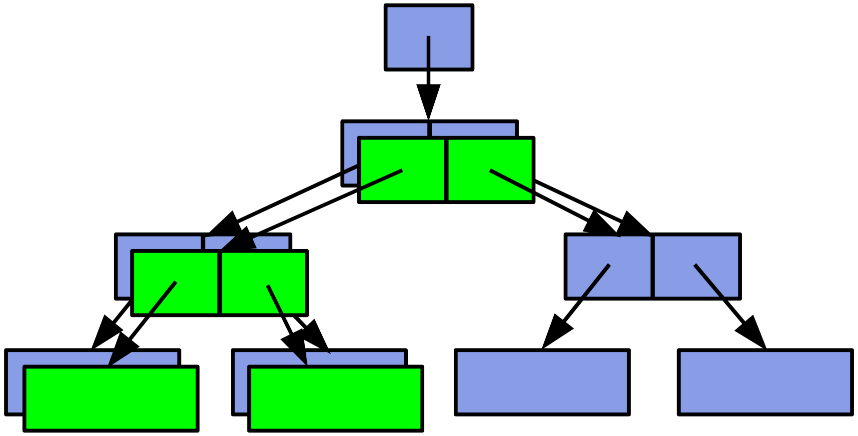
1. Initial block tree



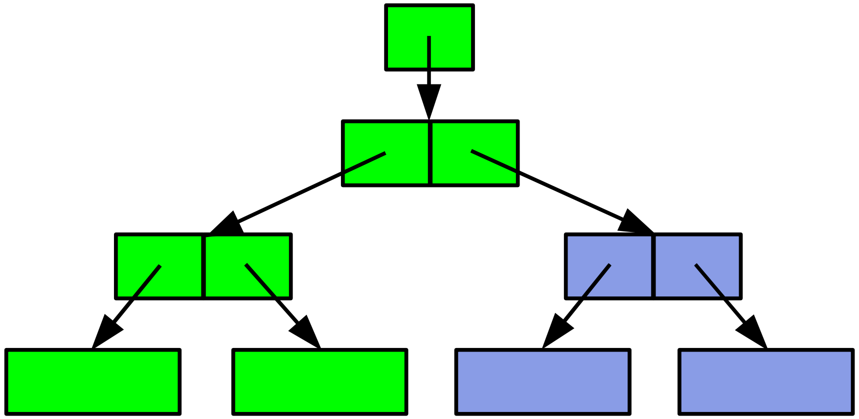
2. COW some blocks



3. COW indirect blocks



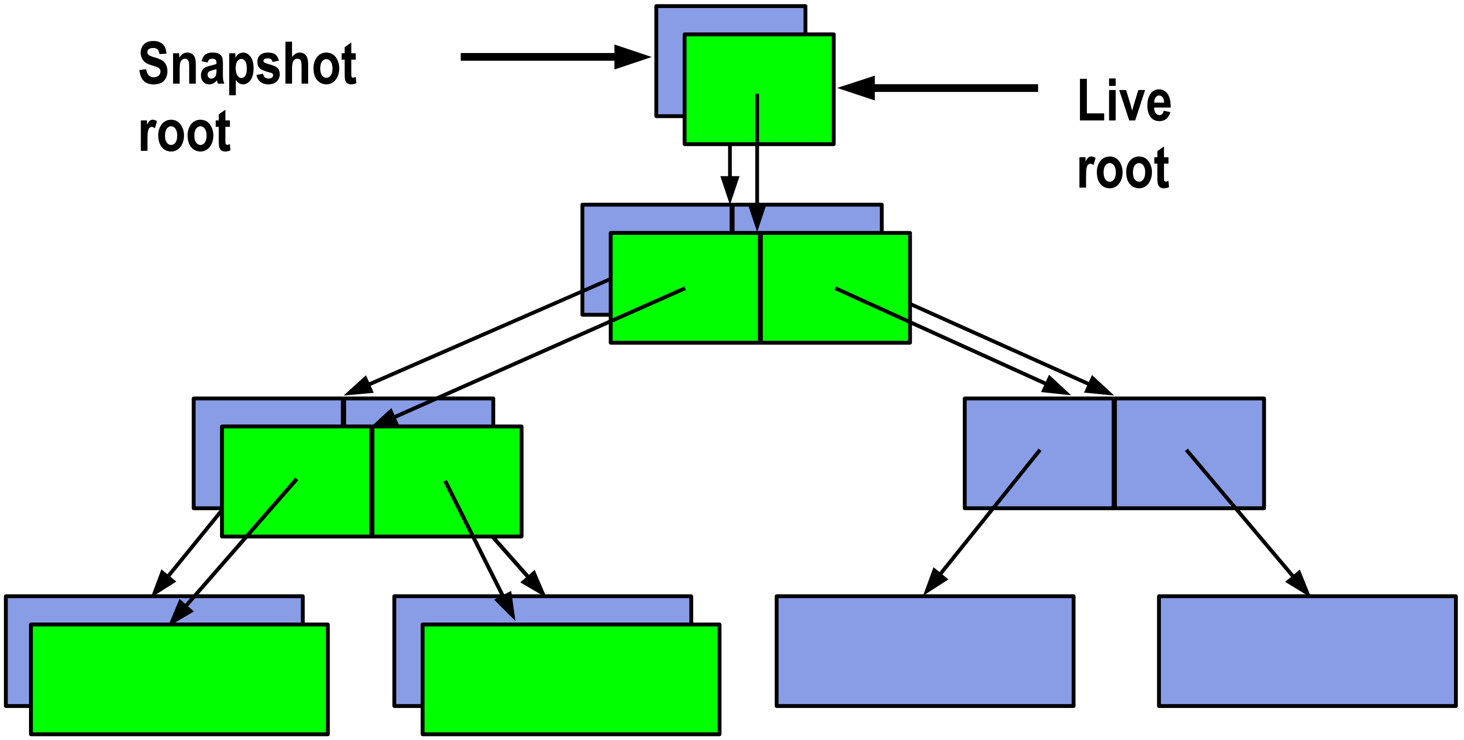
4. Rewrite uberblock (atomic)



Bonus: Constant-Time Snapshots

At end of TX group, don't free COWed blocks

Actually cheaper to take a snapshot than not!



Snapshot root

Live root

End-to-End Data Integrity

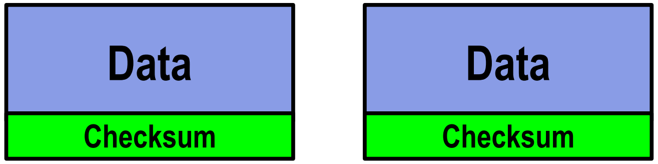
Disk Block Checksums

Checksum stored with data block

Any self-consistent block will pass

Can't even detect stray writes

Inherent FS/volume interface limitation



Disk checksum only validates media

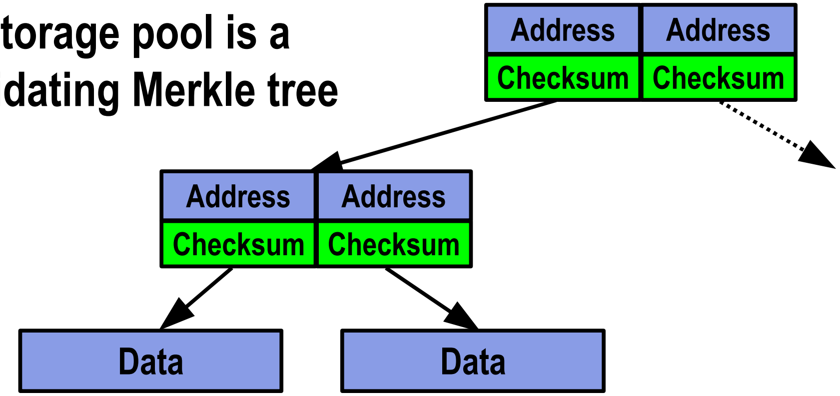
- ✔ Bit rot
- ✗ Phantom writes
- ✗ Misdirected reads and writes
- ✗ DMA parity errors
- ✗ Driver bugs
- ✗ Accidental overwrite

ZFS Data Authentication

Checksum stored in parent block pointer

Fault isolation between data and checksum

Entire storage pool is a
self-validating Merkle tree



ZFS validates the entire I/O path

- ✔ Bit rot
- ✔ Phantom writes
- ✔ Misdirected reads and writes
- ✔ DMA parity errors
- ✔ Driver bugs
- ✔ Accidental overwrite

Disk Scrubbing

Finds latent errors while they're still correctable

- ECC memory scrubbing for disks

Verifies the integrity of all data

- Traverses pool metadata to read every copy of every block
- Verifies each copy against its 256-bit checksum
- Self-healing as it goes

Provides fast and reliable resilvering

- Traditional resync: whole-disk copy, no validity check
- ZFS resilver: live-data copy, everything checksummed
- All data-repair code uses the same reliable mechanism
  - » Mirror/RAID-Z resilver, attach, replace, scrub

ZFS Performance

Copy-on-write design

  - Turns random writes into sequential writes

Multiple block sizes

  - Automatically chosen to match workload

Pipelined I/O

  - Fully scoreboarded I/O pipeline with explicit dependency graphs
  - Priority, deadline scheduling, out-of-order issue, sorting, aggregation

Dynamic striping across all devices

  - Maximizes throughput

Intelligent prefetch

Dynamic Striping

Automatically distributes load across all devices

Writes: striped across all four mirrors

Reads: wherever the data was written

Block allocation policy considers:

Capacity

Performance (latency, BW)

Health (degraded mirrors)

Writes: striped across all five mirrors

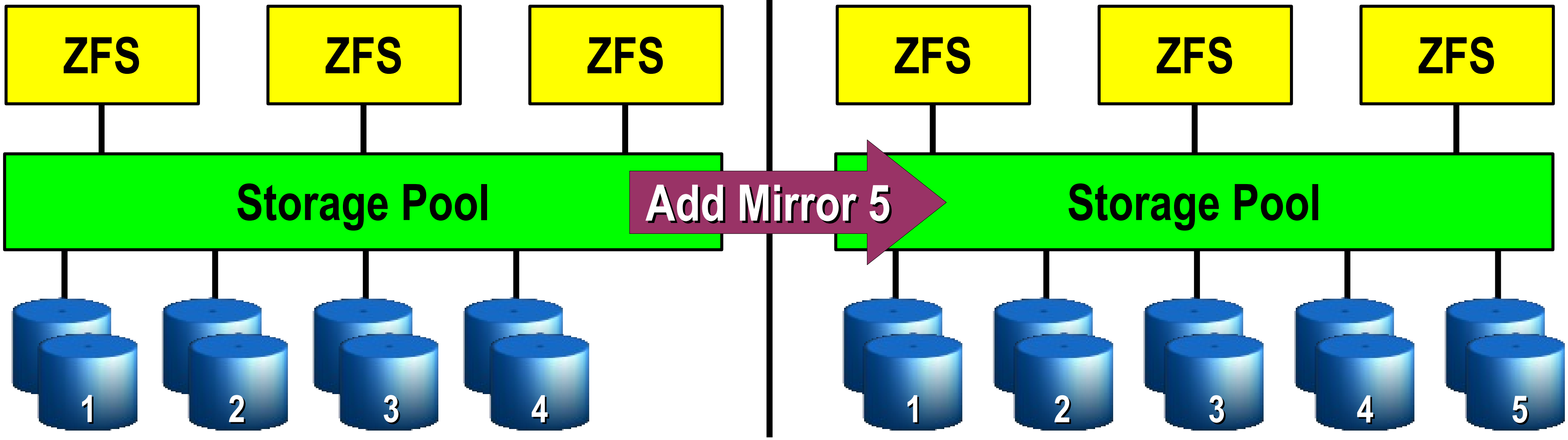
Reads: wherever the data was written

No need to migrate existing data

Old data striped across 1-4

New data striped across 1-5

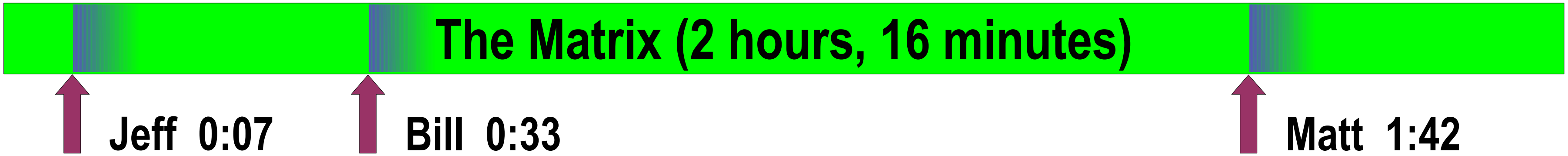
COW gently reallocates old data



Intelligent Prefetch

Multiple independent prefetch streams

- Crucial for any streaming service provider



The Matrix (2 hours, 16 minutes)

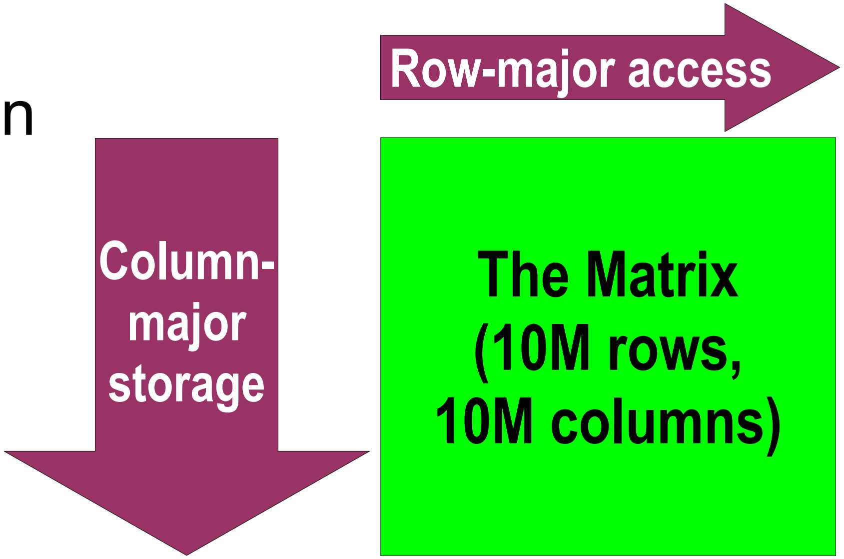
Jeff 0:07

Bill 0:33

Matt 1:42

Automatic length and stride detection

- Great for HPC applications
- ZFS understands the matrix multiply problem
  - » Detects any linear access pattern
  - » Forward or backward



Row-major access

Column-major storage

The Matrix (10M rows, 10M columns)

ZFS Administration

Pooled storage – no more volumes!

- All storage is shared – no wasted space, no wasted bandwidth

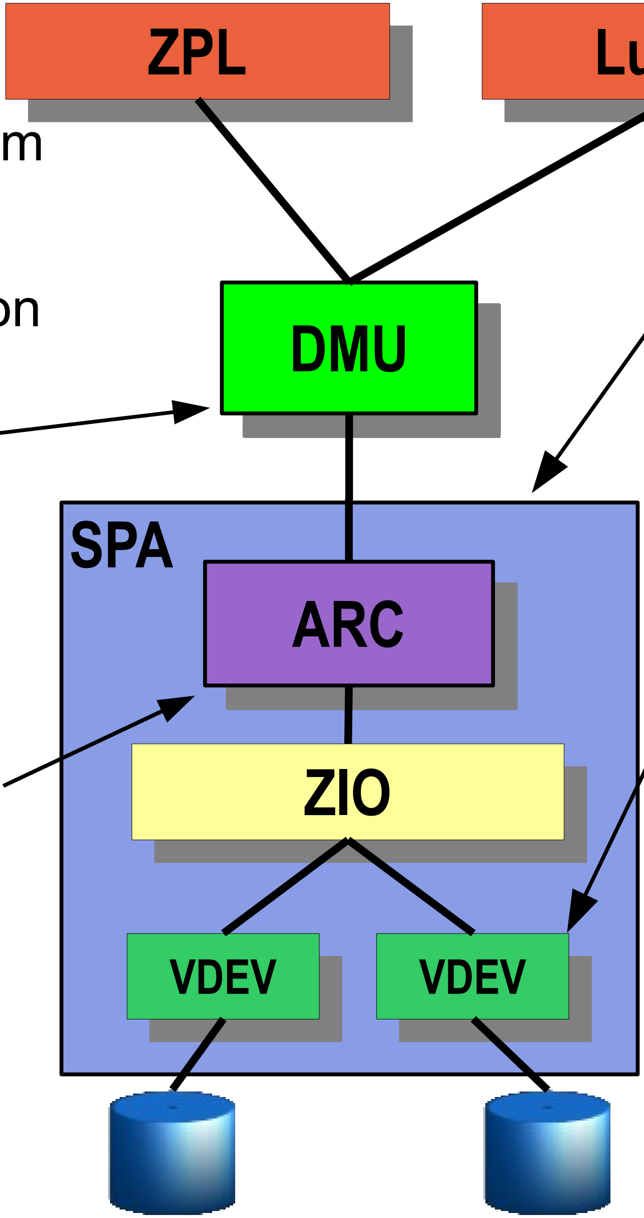
Hierarchical filesystems with inherited properties

- Filesystems become administrative control points
  - » Per-dataset policy: snapshots, compression, backups, privileges, etc.
  - » Who's using all the space? du(1) takes forever, but df(1M) is instant!

- Manage logically related filesystems as a group
- Control compression, checksums, quotas, reservations, and more
- Mount and share filesystems without /etc/vfstab or /etc/dfs/dfstab
- Inheritance makes large-scale administration a snap

Online everything

ZFS IO Stack



POSIX Interface

Look and feel of a file system but much, much, more

VNODE/VFS Implementation

Object-Based Transactions

“Make these 7 changes to these 3 objects”

All-or-nothing

Transaction Group Batch I/O

Schedule, aggregate, and issue I/O at will

No resync if power lost

Runs at platter speed

Transaction Group Commit

Again, all-or-nothing

Always consistent on disk

No journal – not needed

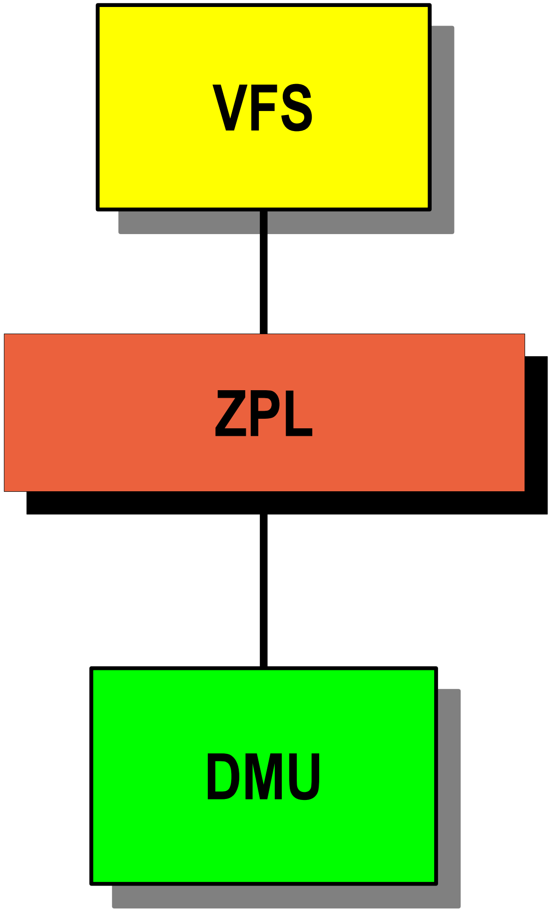
Virtual Devices

Take the block interface to the lowest level

Provide mirroring, raid-z, and spare operations

Dirty Time Logging for quick resilvering

ZPL (ZFS POSIX Layer)

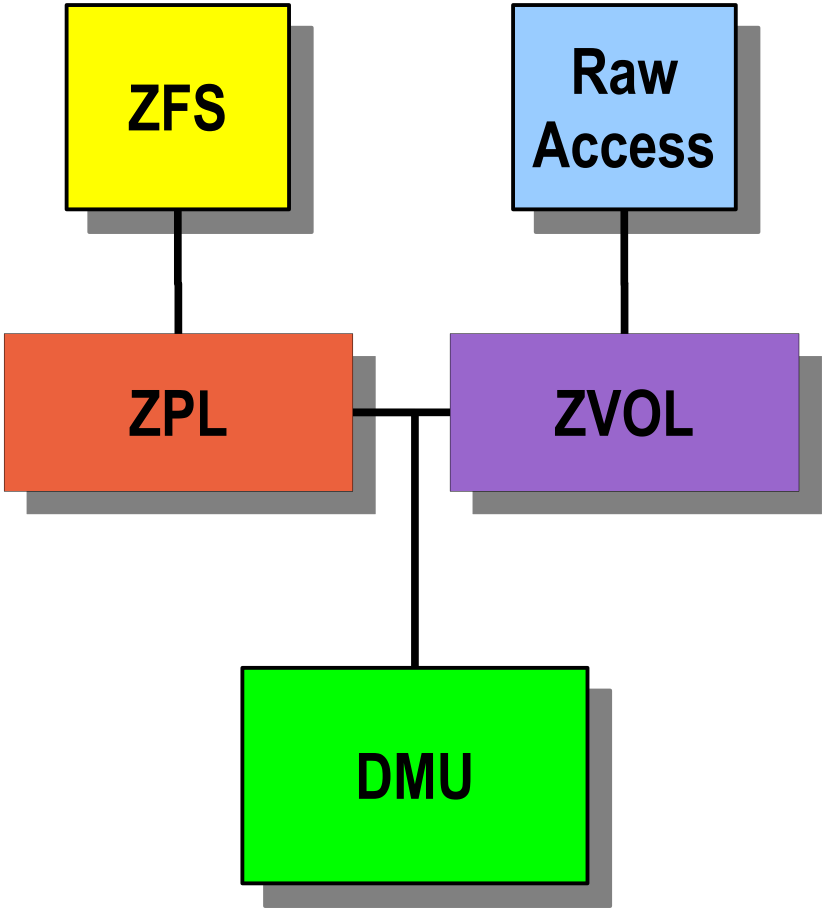


ZPL is the primary interface for interacting with ZFS as a filesystem.

It is a layer that sits atop the DMU and presents a filesystem abstraction of files and directories.

It is responsible for bridging the gap between the VFS interfaces and the underlying DMU interfaces.

ZVOL (ZFS Emulated Volume)



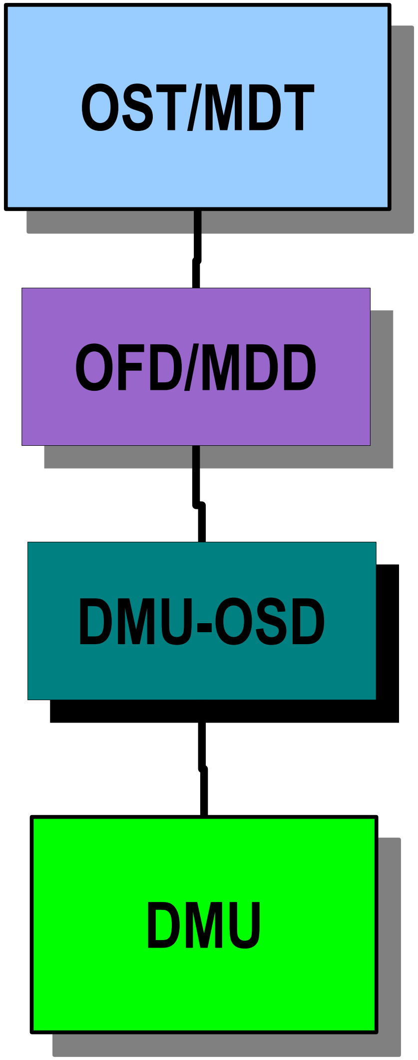
Provides a mechanism for creating logical volumes which can be used as block or character devices

Can be used to create sparse volumes (aka “thin provisioning”)

Ability to specify the desired blocksize

Storage is backed by storage pool

Lustre Object Storage Device (OSD)



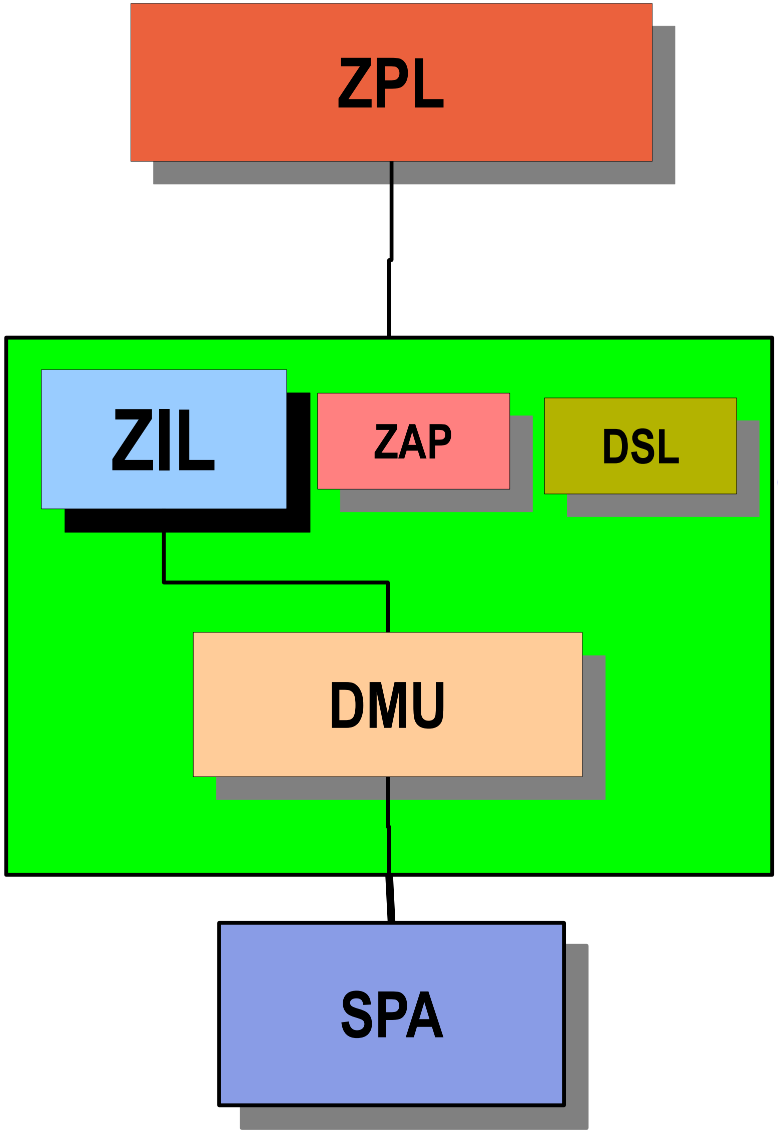
Provides object-based storage like ldiskfs OSD

Lets Lustre scale for next generation systems

Allows Lustre to utilize advanced features of ZFS

Storage is backed by storage pool

ZIL (ZFS Intent Log)



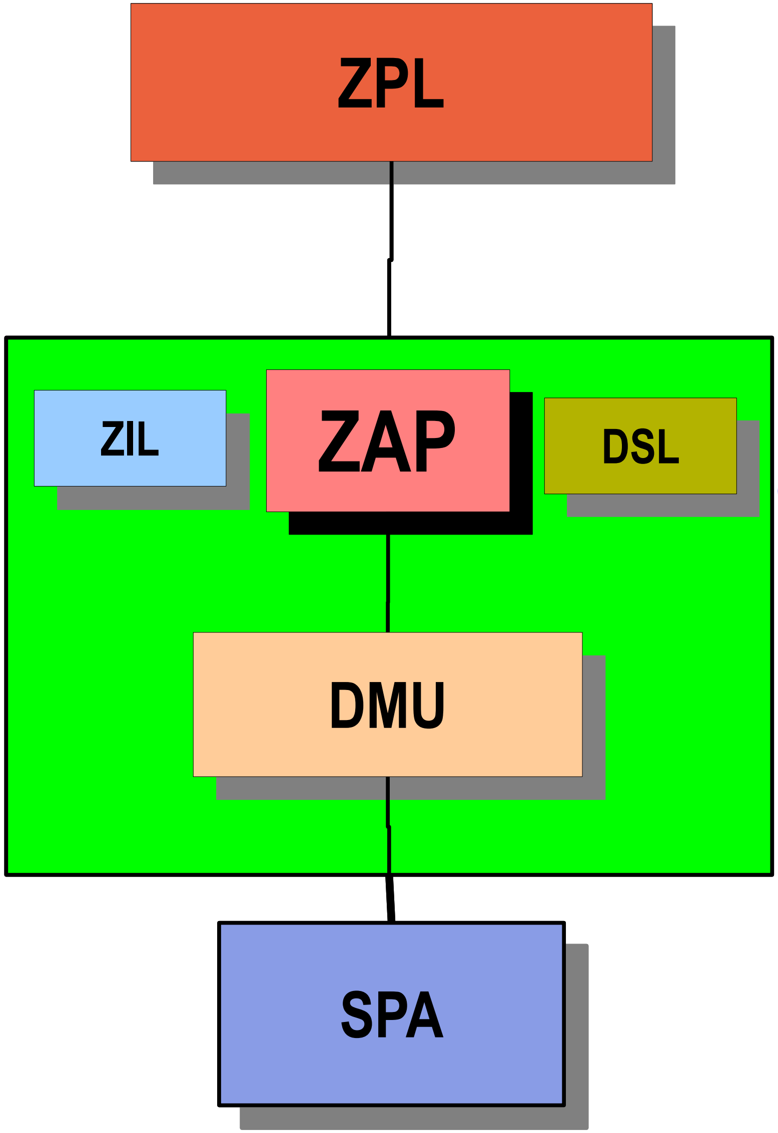
Per-dataset transaction log which can be replayed upon a system crash

Provides semantics to guarantee data is on disk when the write(2), read(2), fsync(3C) syscall returns

Allows operation consistency without the need for expensive transaction commit operation

Used when applications specify (O_DSYNC) or fsync(3C) issued

ZAP (ZFS Attribute Processor)



Makes arbitrary {key, value} associations within an object

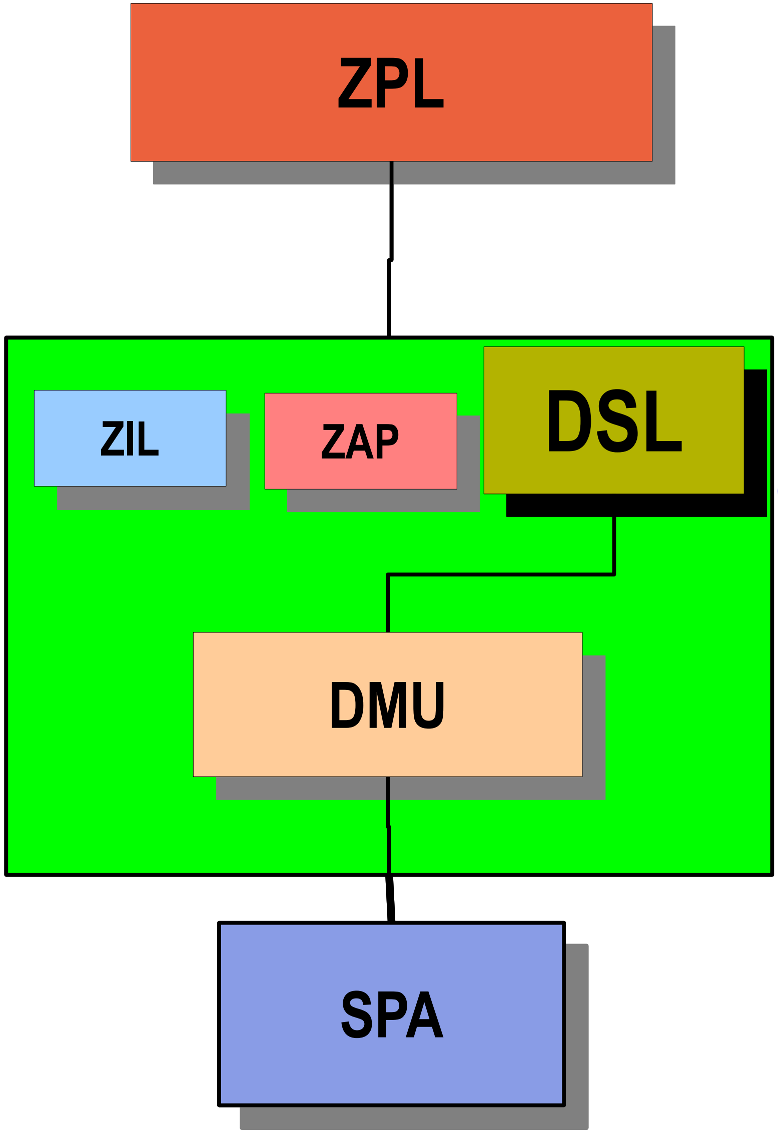
Commonly used to implement directories within the ZPL

Pool-wide properties storage

MicroZAP – when number of entries is relatively small

fatZAP - used for larger directories, long keys, or values other than uint64_t

DSL (Dataset and Snapshot Layer)



Aggregates DMU objects in a hierarchical namespace

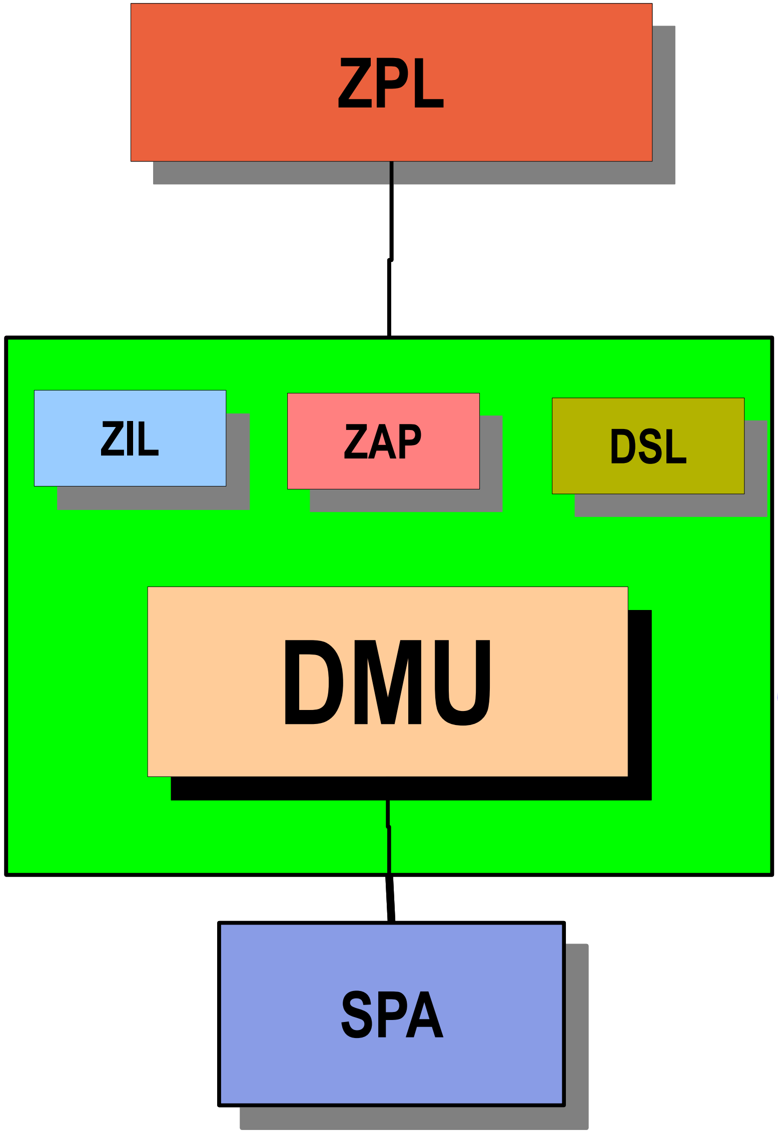
Allows inheriting properties, as well as quota and reservation enforcement

Describes types of object sets

- ZFS Filesystems
- Clones
- ZFS Volumes
- Snapshots

Manages snapshots and clones of object sets

DMU (Data Management Unit)



Responsible for presenting a transactional object model, built atop the flat address space presented by the SPA.

Consumers interact with the DMU via object sets, objects, and transactions.

An object set is a collection of objects, where each object are pieces of storage from the SPA (i.e. a collection of blocks).

Each transaction is a series of operations that must be committed to disk as a group; it is central to the on-disk consistency for ZFS.

Universal Storage

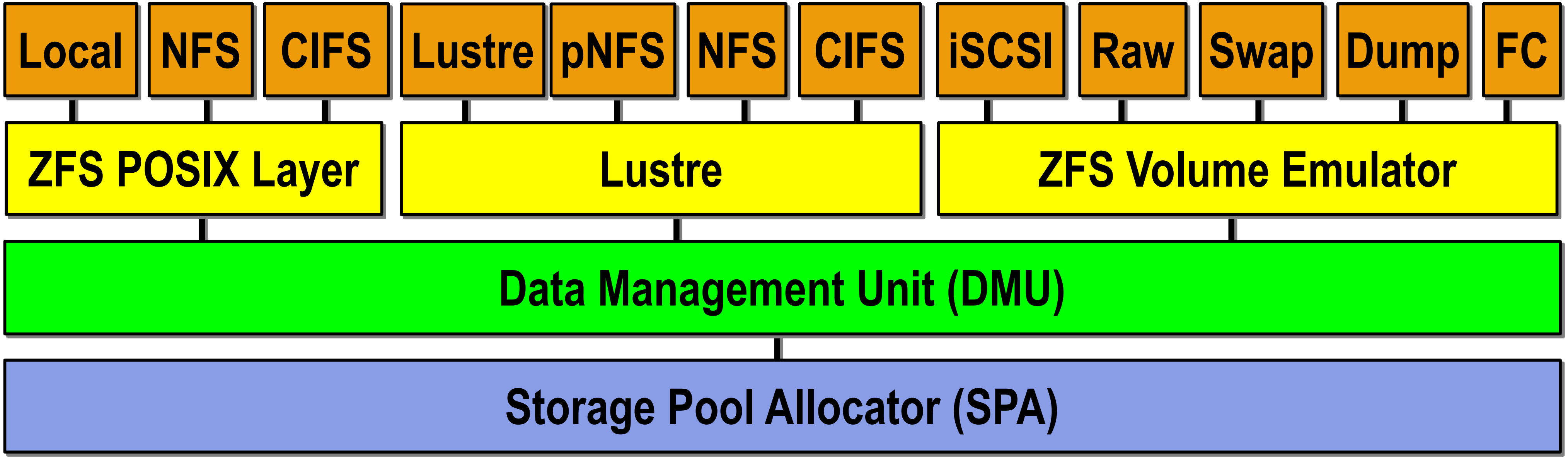
DMU is a general-purpose transactional object store

- ZFS dataset = up to 2^{48} objects, each up to 2^{64} bytes

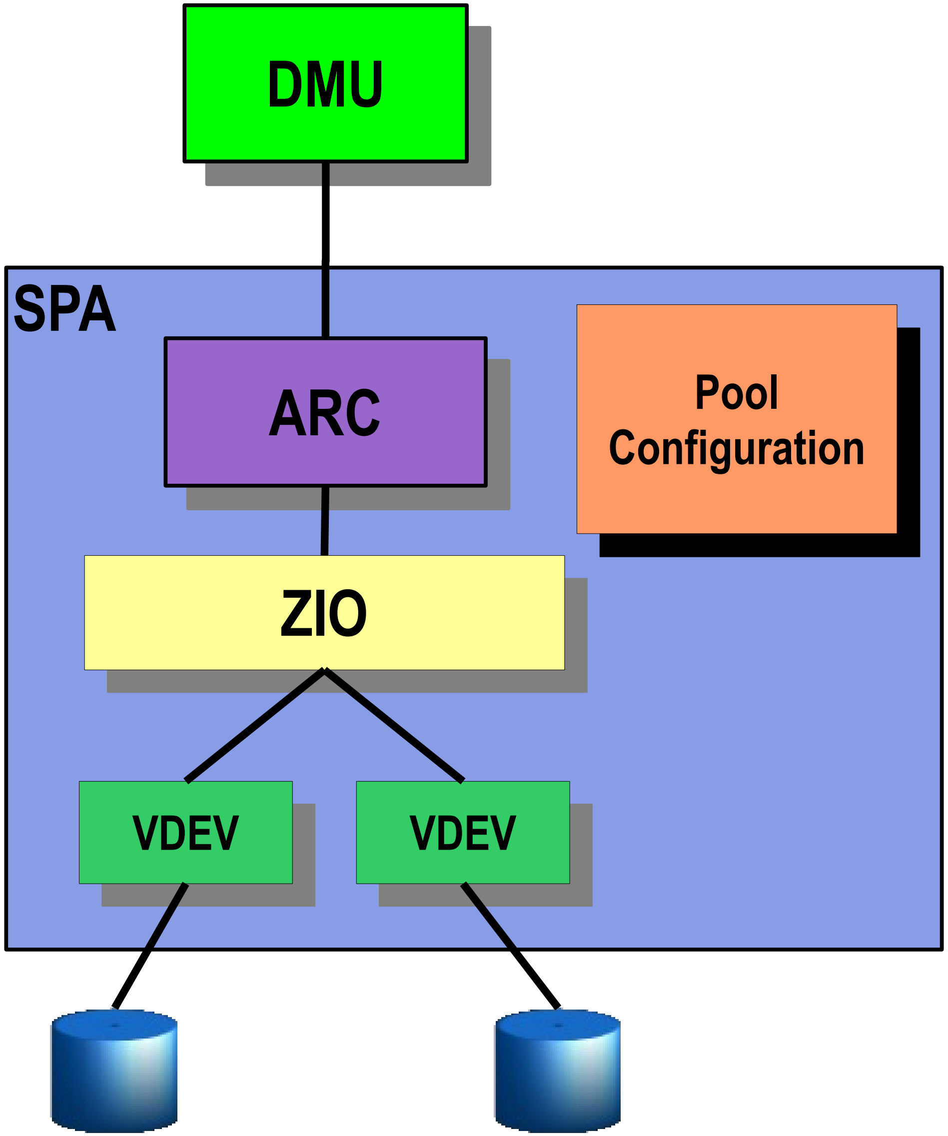
Key features common to all datasets

- Snapshots, compression, encryption, de-duplication, end-to-end data integrity

Any flavor you want: file, block, object, network



Pool Configuration



Provides public interfaces to manipulate pool configuration

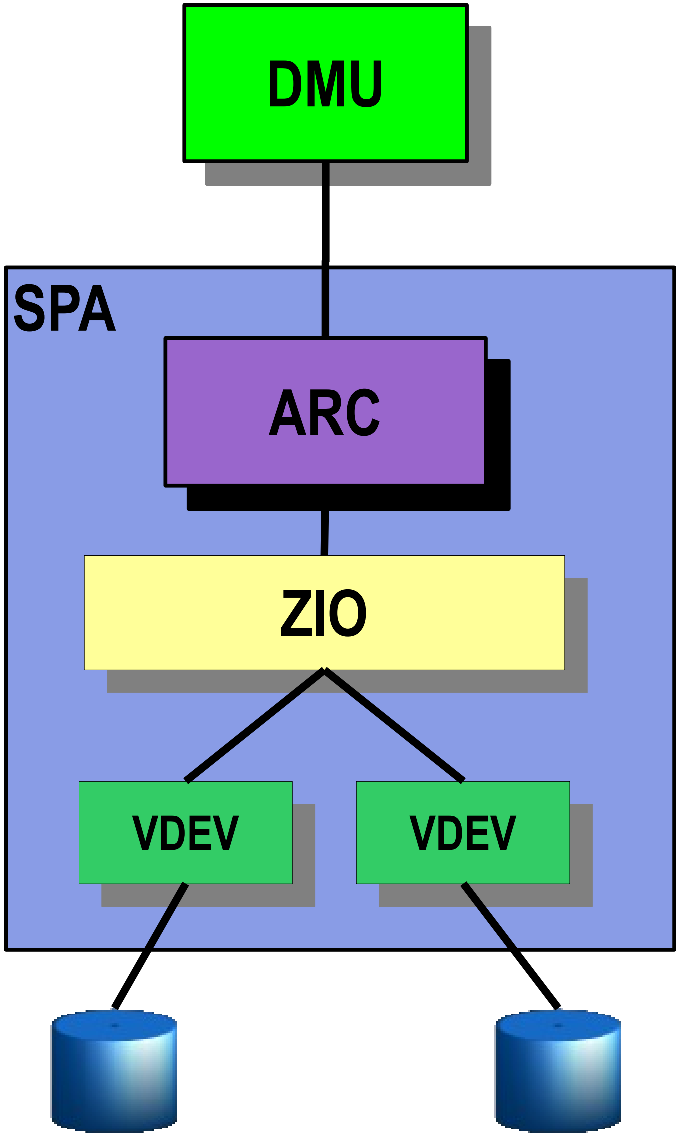
Interface can create, destroy, import, export, and pools

Glues ZIO and vdev layers into a consistent pool object

Manages Pool Namespace

Enables periodic data sync

ARC (Adaptive Replacement Cache)



DVA (Data Virtual Address) based cache used by DMU

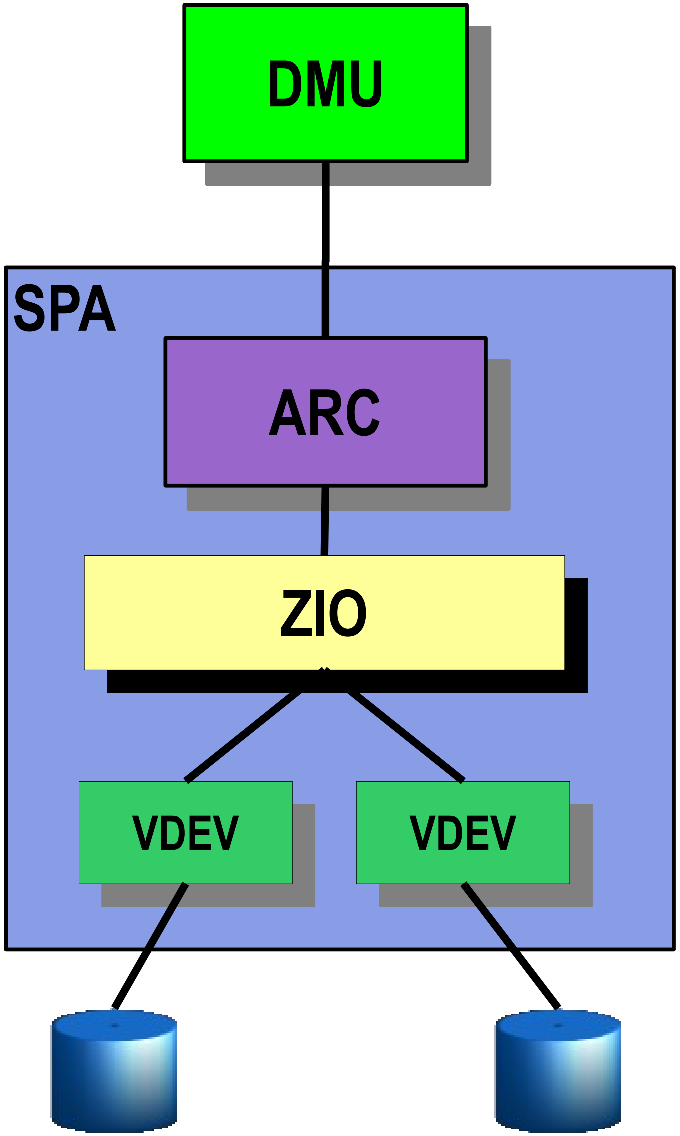
Self-tuning cache will adjust based on I/O workload

Replaces the page cache

Central point for memory management for the SPA

Ability to evict buffers as a result of memory pressure

ZIO (ZFS I/O Pipeline)



Centralized I/O framework

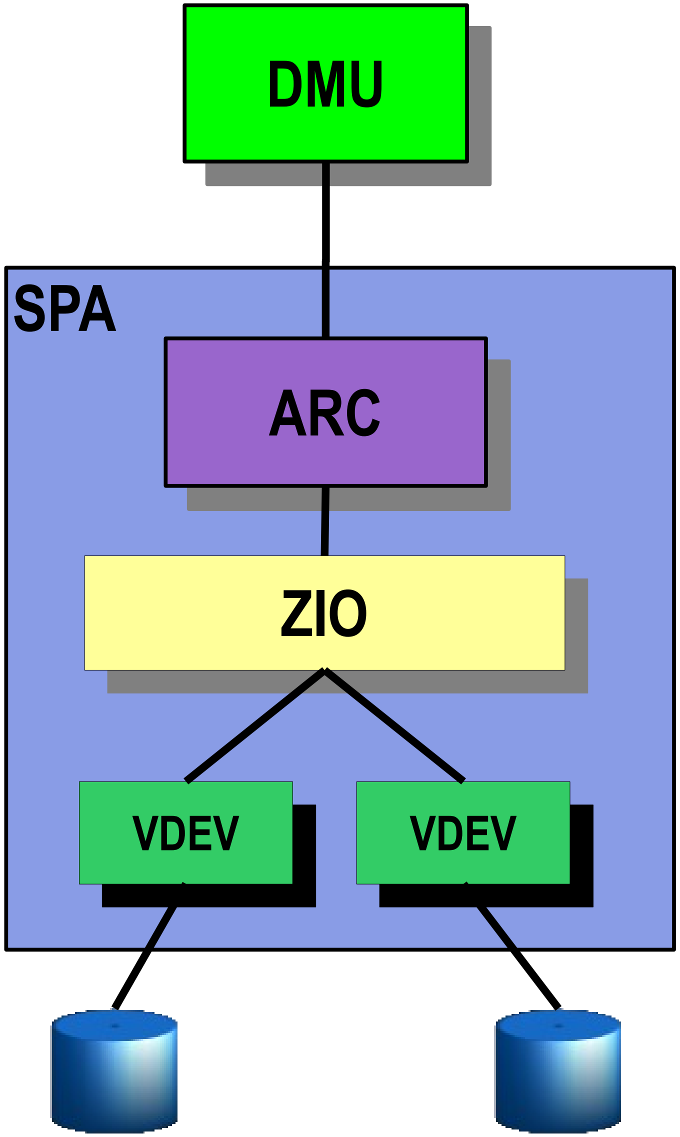
I/Os follow a structured pipeline

Translates DVAs to logical locations on vdevs

Drives dynamic striping and I/O retries across all active vdevs

Drives compression, checksums, data redundancy

VDEV (Virtual Devices)



Abstraction of devices

Physical devices (leaf vdevs)

Logical devices (internal vdevs)

Implements data replication

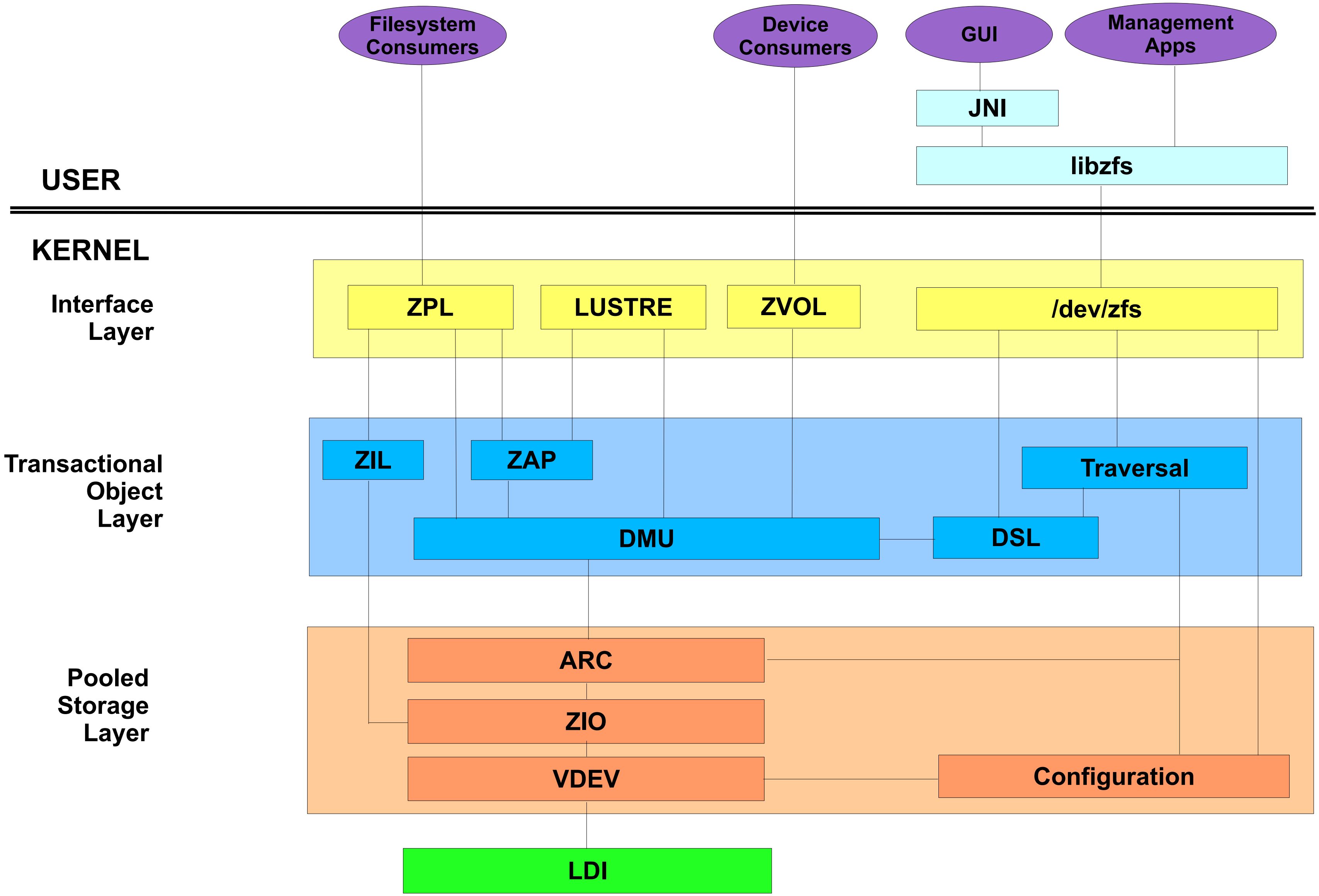
Mirroring, RAID-Z, RAID-Z2

Interface with block devices

Provides I/O scheduling

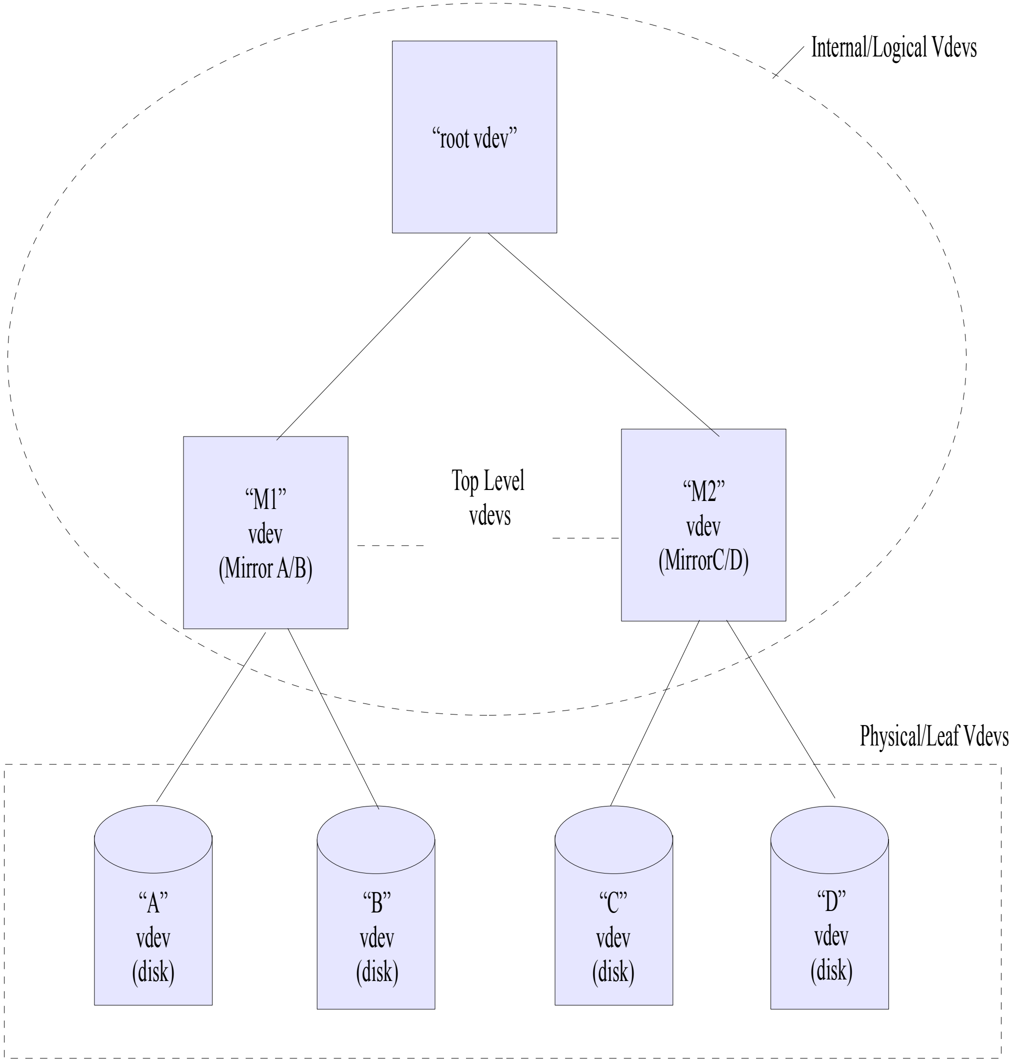
Controls device cache flush

Source Overview



On-Disk Format Review

Virtual Devices



ZFS Storage pools are made up of a collection of virtual devices stored in a tree structure

leaf vdevs (physical devices)

logical vdevs (mirrors, raidz, etc.)

All pools contain a special logical vdev called the "root" vdev

Direct children of the "root" vdev are called top-level vdevs

VDEV Labels

Four copies of the VDEV label are written to each physical VDEV

Two labels at the front of the device and two at the end

Labels are identical for all VDEVs within the pool

Label Details

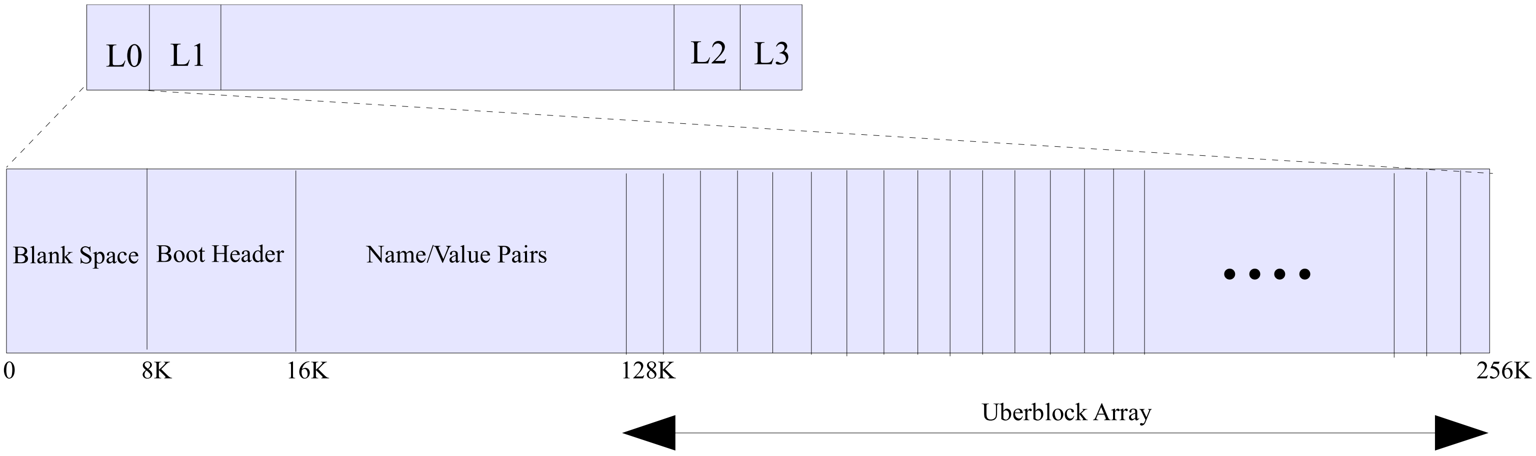
Each VDEV label consists of:

8KB blank space (support for VTOC disk label)

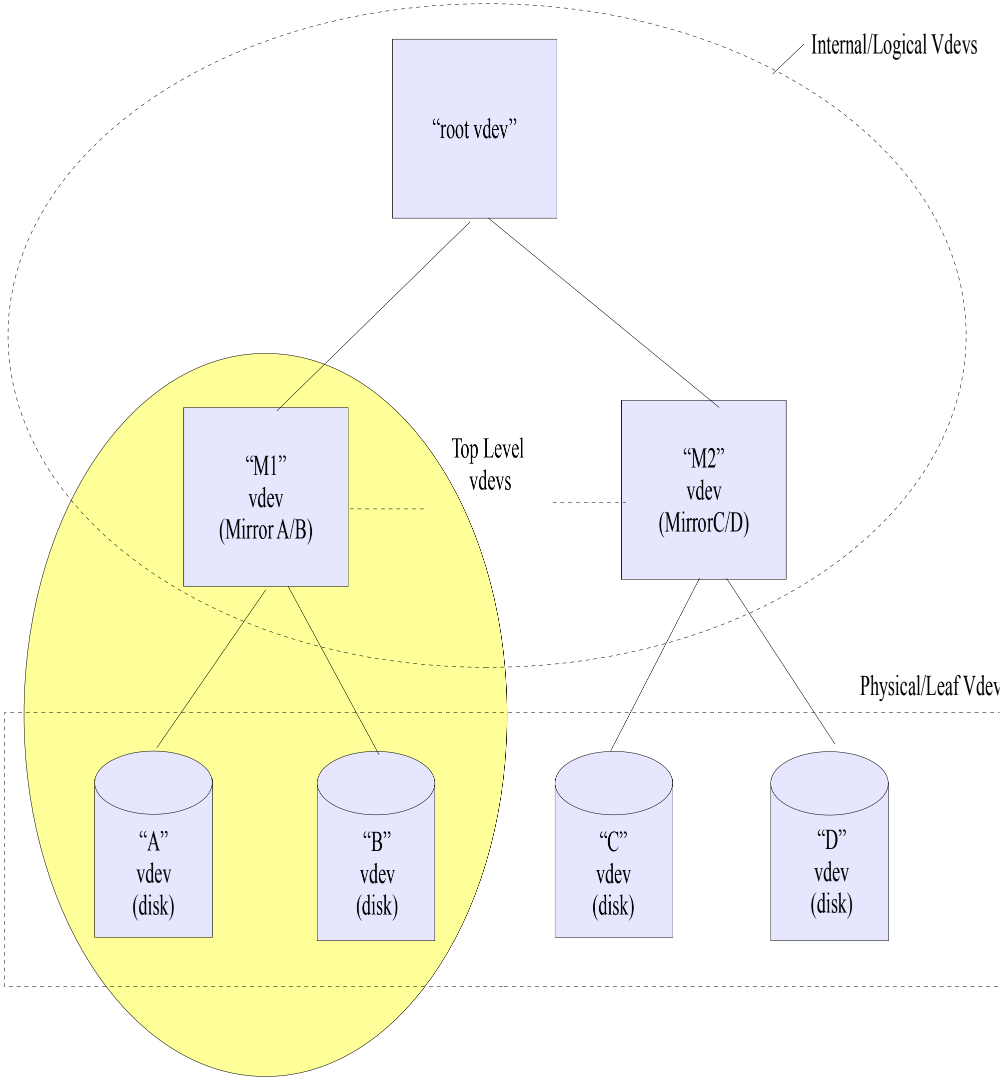
8KB for the boot header (future)

112KB of name-value pairs

128KB of 1K sized uberblock structures (ring buffer)



VDEV Trees



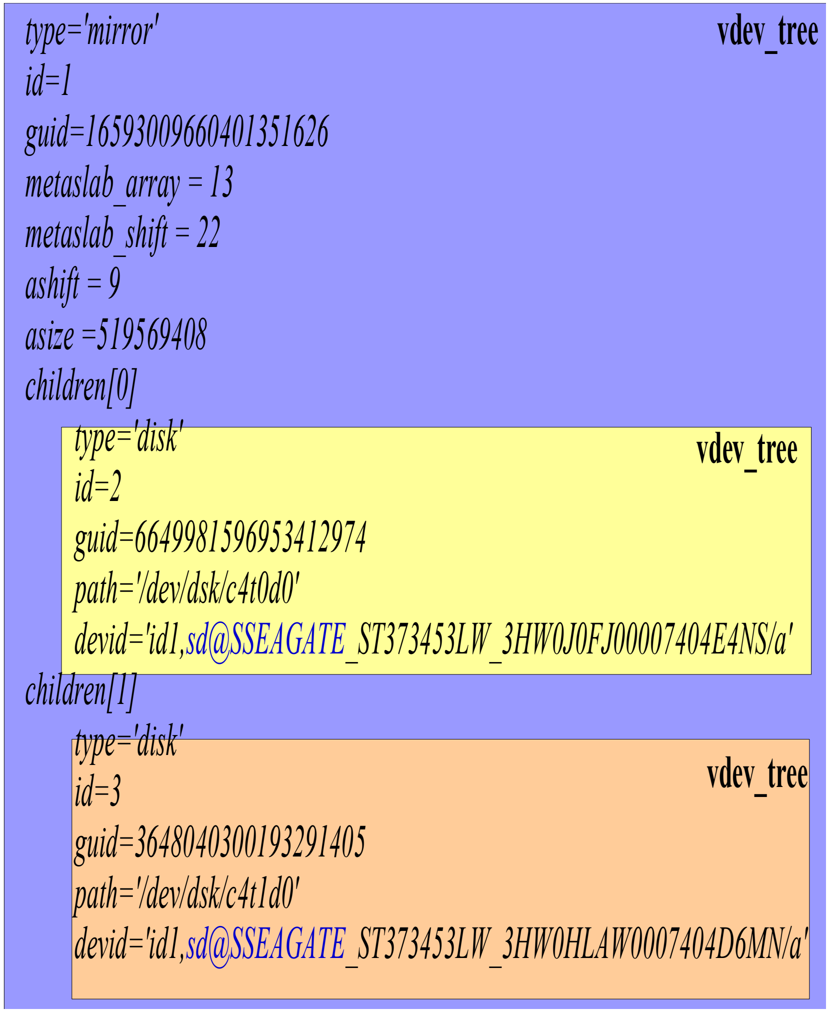
The 112KB used for the NVlist contains information that describes all the related VDEVs

Related VDEVs are VDEVs that are rooted at a common top-level VDEVs

NVlist VDEV View

One of the name-value pairs stored in the label is the VDEV tree

The VDEV tree recursively describes the hierarchical view of the related VDEV

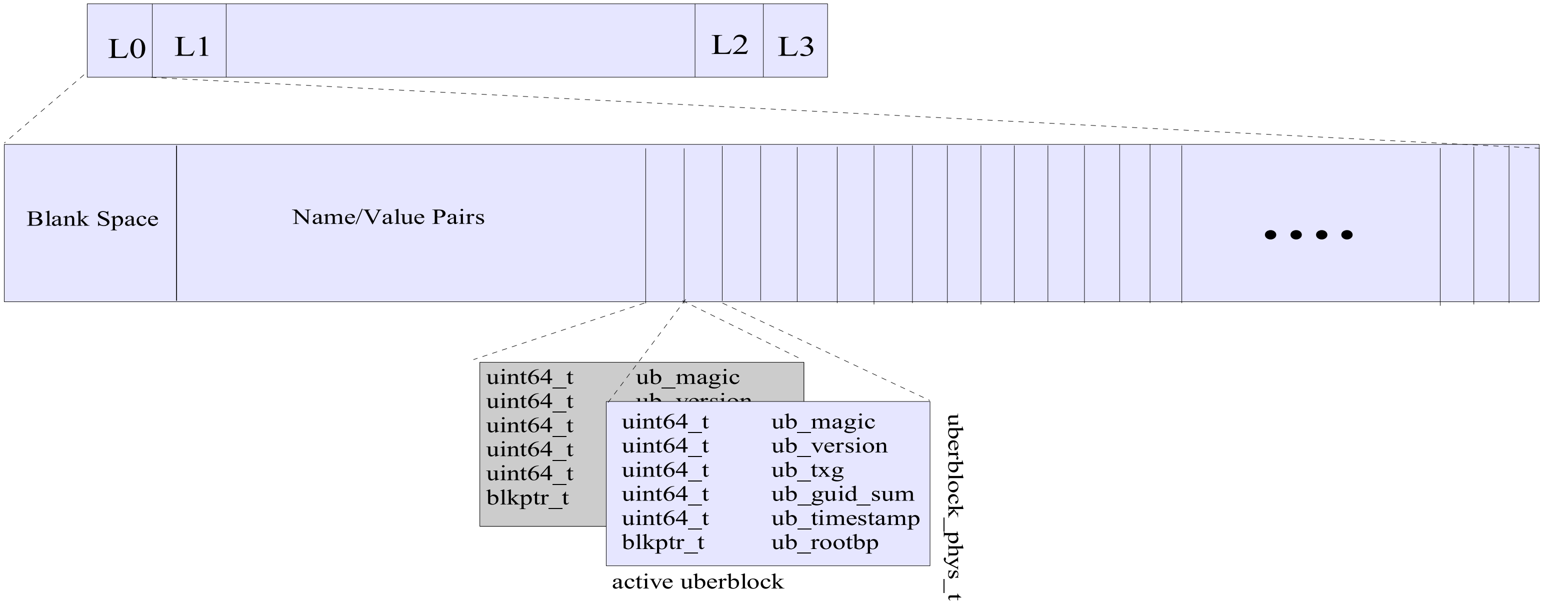


```
type='mirror'                                   vdev_tree
id=1
guid=16593009660401351626
metaslab_array = 13
metaslab_shift = 22
ashift = 9
asize =519569408
children[0]
    type='disk'                                 vdev_tree
    id=2
    guid=6649981596953412974
    path='/dev/dsk/c4t0d0'
    devid='id1,sd@SSEAGATE_ST373453LW_3HW0J0FJ00007404E4NS/a'
children[1]
    type='disk'
    id=3                                        vdev_tree
    guid=3648040300193291405
    path='/dev/dsk/c4t1d0'
    devid='id1,sd@SSEAGATE_ST373453LW_3HW0HLAW0007404D6MN/a'
```

Uberblock

The VDEV label contains an array of uberblocks (128KB)

The uberblock with the highest transaction group and contains a valid SHA-256 checksum is the active uberblock



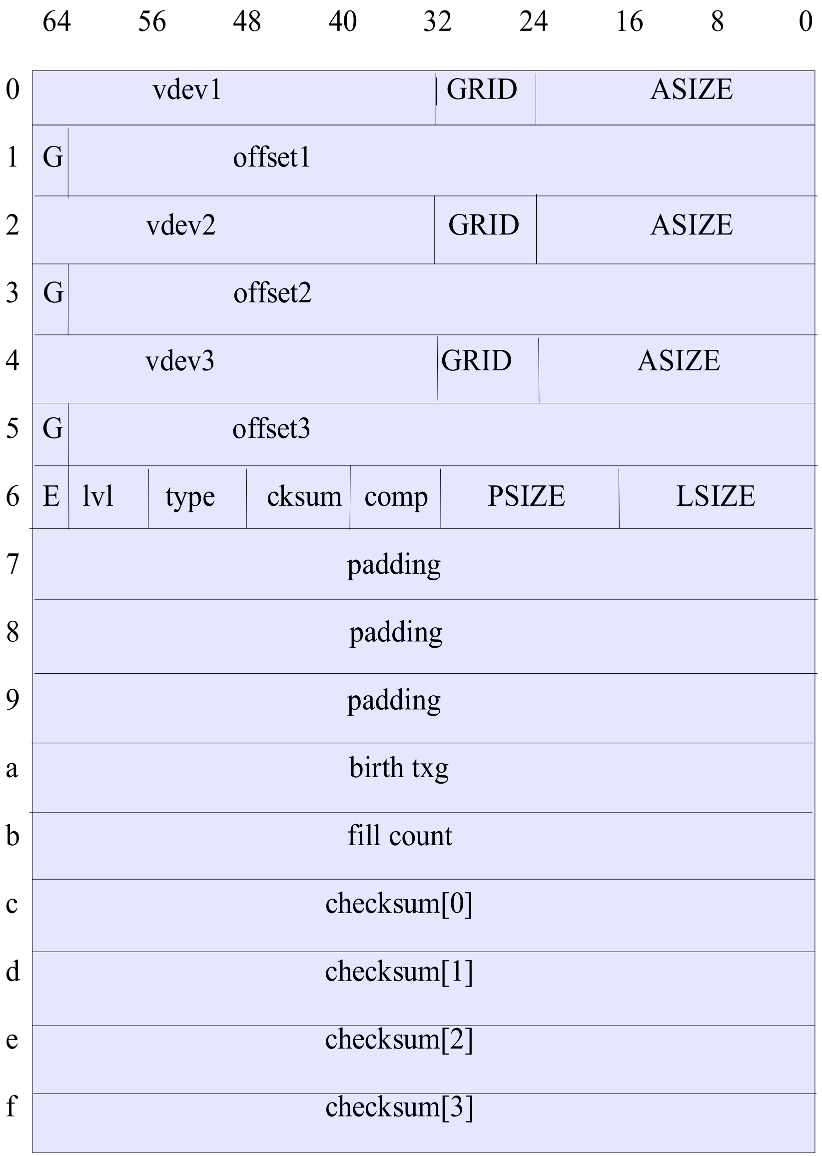
Block Pointers

Block pointers are 128 byte structured used to physically locate, verify, and describe data on disk

Each block pointer contains a DVA (Data Virtual Address) used to address the data

Comprised of VDEV, offset

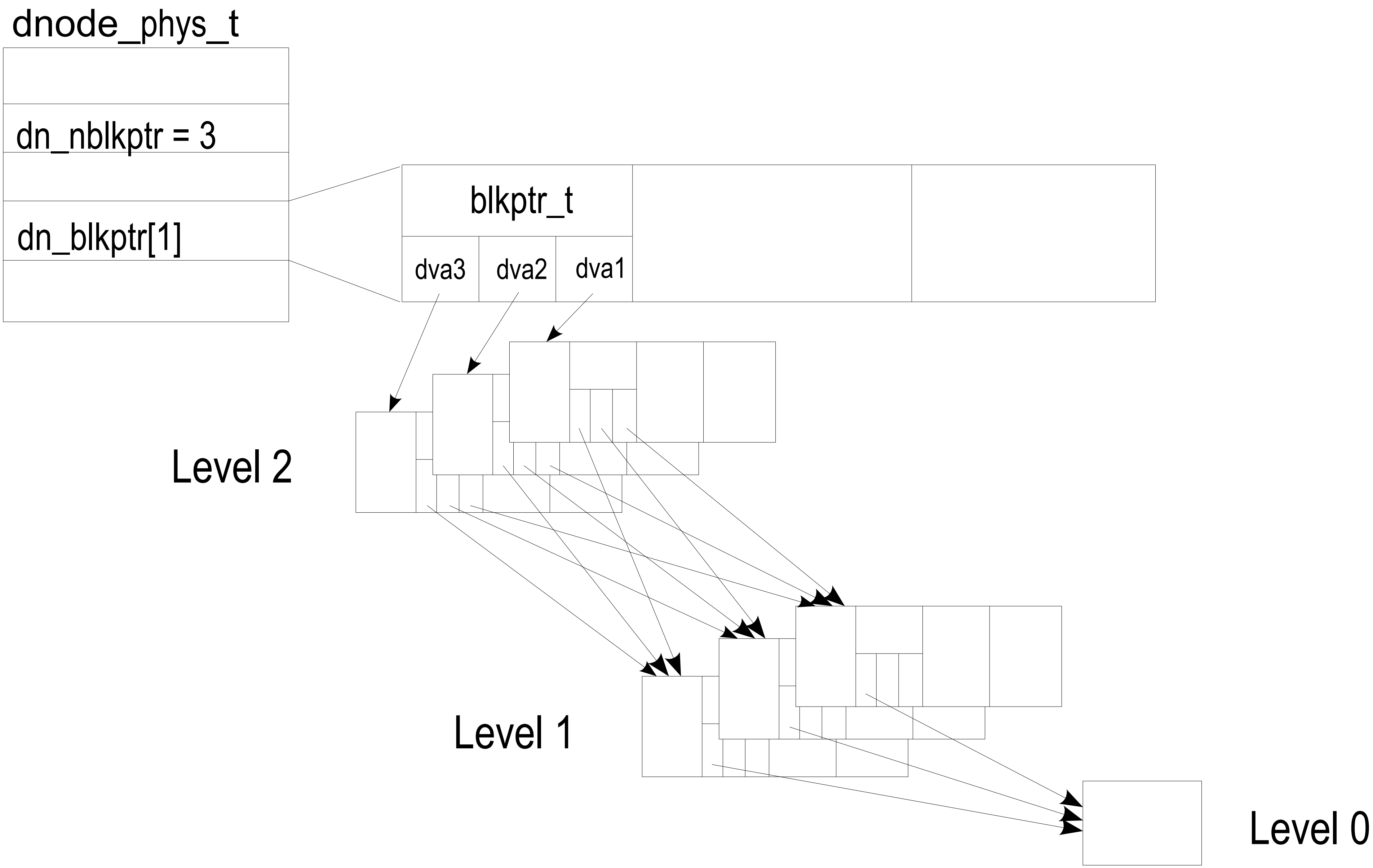
Multiple DVAs give multiple paths to the data block



64 56 48 40 32 24 16 8 0

<table>
<tr><td>0</td><td colspan="5">vdev1</td><td>GRID</td><td colspan="3">ASIZE</td></tr>
<tr><td>1</td><td>G</td><td colspan="8">offset1</td></tr>
<tr><td>2</td><td colspan="5">vdev2</td><td>GRID</td><td colspan="3">ASIZE</td></tr>
<tr><td>3</td><td>G</td><td colspan="8">offset2</td></tr>
<tr><td>4</td><td colspan="5">vdev3</td><td>GRID</td><td colspan="3">ASIZE</td></tr>
<tr><td>5</td><td>G</td><td colspan="8">offset3</td></tr>
<tr><td>6</td><td>E</td><td>lvl</td><td>type</td><td>cksum</td><td>comp</td><td colspan="2">PSIZE</td><td colspan="2">LSIZE</td></tr>
<tr><td>7</td><td colspan="9">padding</td></tr>
<tr><td>8</td><td colspan="9">padding</td></tr>
<tr><td>9</td><td colspan="9">padding</td></tr>
<tr><td>a</td><td colspan="9">birth txg</td></tr>
<tr><td>b</td><td colspan="9">fill count</td></tr>
<tr><td>c</td><td colspan="9">checksum[0]</td></tr>
<tr><td>d</td><td colspan="9">checksum[1]</td></tr>
<tr><td>e</td><td colspan="9">checksum[2]</td></tr>
<tr><td>f</td><td colspan="9">checksum[3]</td></tr>
</table>

Ditto Blocks

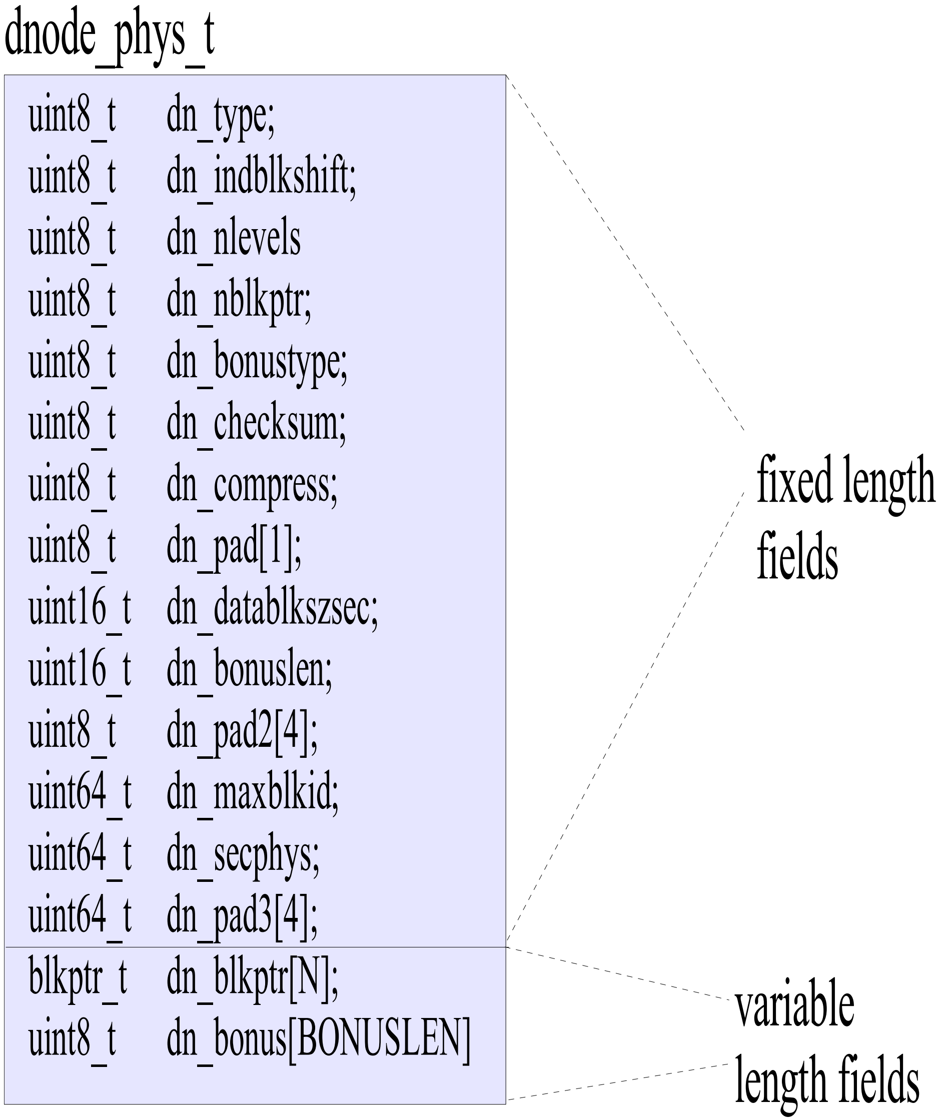


Dnode

dnodes are 512-byte structures which organize a collection of blocks which make up an object

The portion of the dnode which we store on-disk is pictured here

The dn_blkptr will point to the array of block pointer which will point to the indirect, direct, and data blocks



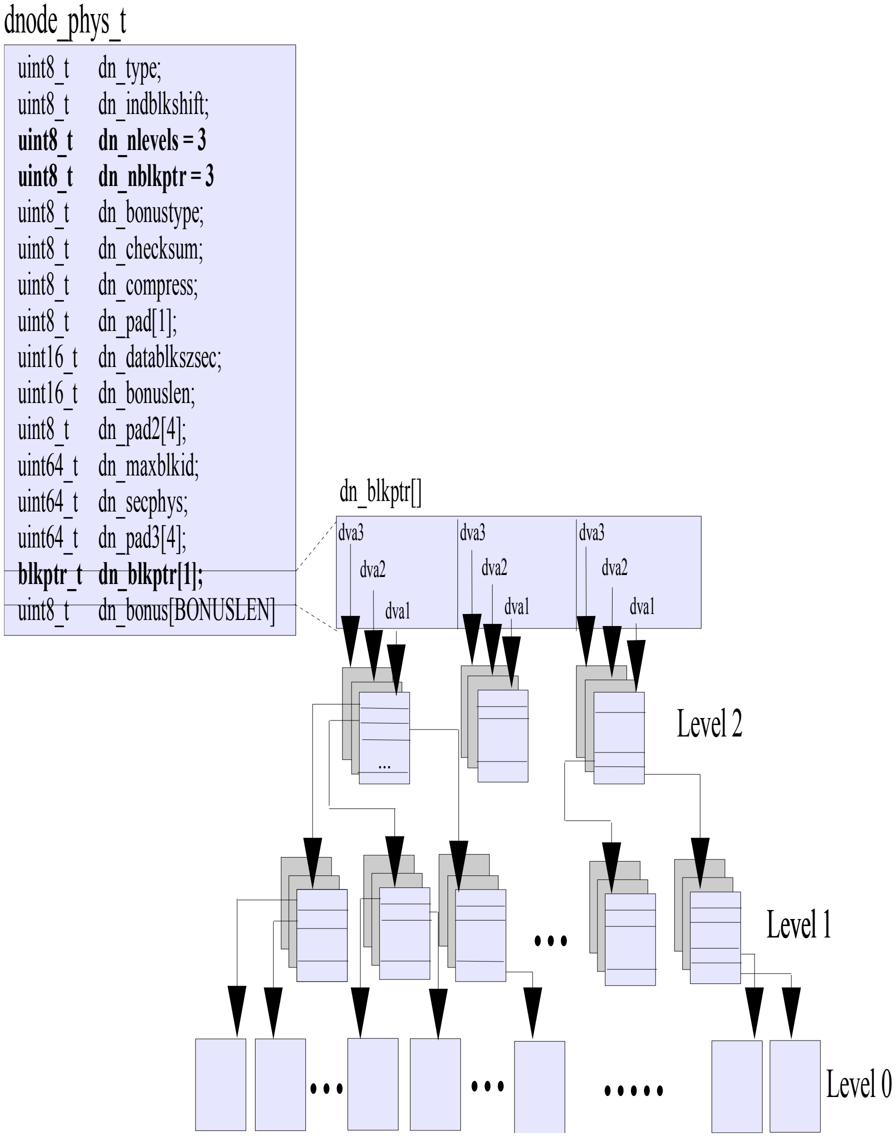
dnode_phys_t

```
uint8_t   dn_type;
uint8_t   dn_indblkshift;
uint8_t   dn_nlevels
uint8_t   dn_nblkptr;
uint8_t   dn_bonustype;
uint8_t   dn_checksum;
uint8_t   dn_compress;
uint8_t   dn_pad[1];
uint16_t  dn_datablkszsec;
uint16_t  dn_bonuslen;
uint8_t   dn_pad2[4];
uint64_t  dn_maxblkid;
uint64_t  dn_secphys;
uint64_t  dn_pad3[4];
blkptr_t  dn_blkptr[N];
uint8_t   dn_bonus[BONUSLEN]
```

fixed length fields

variable length fields

Indirect Blocks



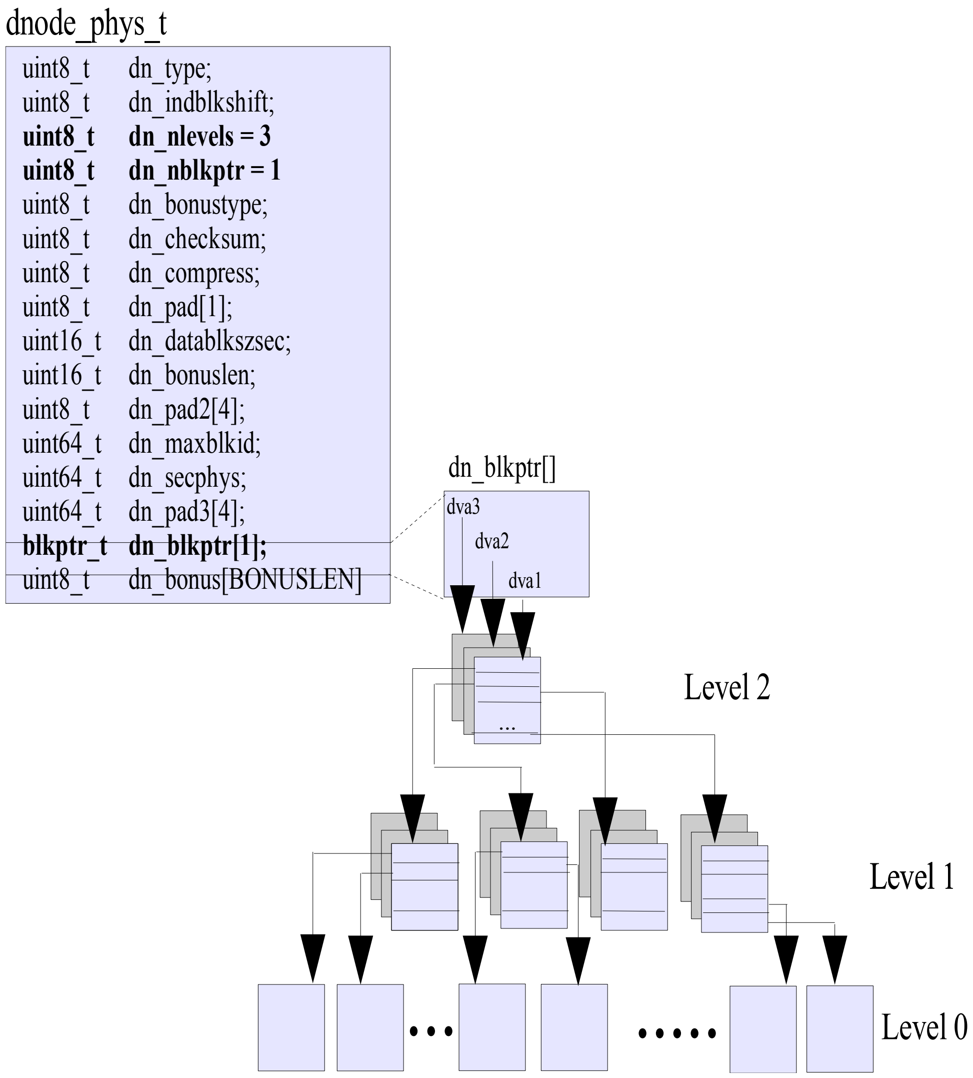
```
dnode_phys_t
uint8_t   dn_type;
uint8_t   dn_indblkshift;
uint8_t   dn_nlevels = 3
uint8_t   dn_nblkptr = 3
uint8_t   dn_bonustype;
uint8_t   dn_checksum;
uint8_t   dn_compress;
uint8_t   dn_pad[1];
uint16_t  dn_datablkszsec;
uint16_t  dn_bonuslen;
uint8_t   dn_pad2[4];
uint64_t  dn_maxblkid;
uint64_t  dn_secphys;
uint64_t  dn_pad3[4];
blkptr_t  dn_blkptr[1];
uint8_t   dn_bonus[BONUSLEN]
```

Each dnode_phys_t structure has up to 3 block pointers

The largest indirect block size (128KB) can contain 1024 block pointers

E.g.: Assume 128Kb blocks, each indirection level gives 1024 block pointers * 3 (3072) and addresses a 384MB file

Indirect Blocks (cont.)



dnode_phys_t

```
uint8_t    dn_type;
uint8_t    dn_indblkshift;
uint8_t    dn_nlevels = 3
uint8_t    dn_nblkptr = 1
uint8_t    dn_bonustype;
uint8_t    dn_checksum;
uint8_t    dn_compress;
uint8_t    dn_pad[1];
uint16_t   dn_datablkszsec;
uint16_t   dn_bonuslen;
uint8_t    dn_pad2[4];
uint64_t   dn_maxblkid;
uint64_t   dn_secphys;
uint64_t   dn_pad3[4];
blkptr_t   dn_blkptr[1];
uint8_t    dn_bonus[BONUSLEN]
```

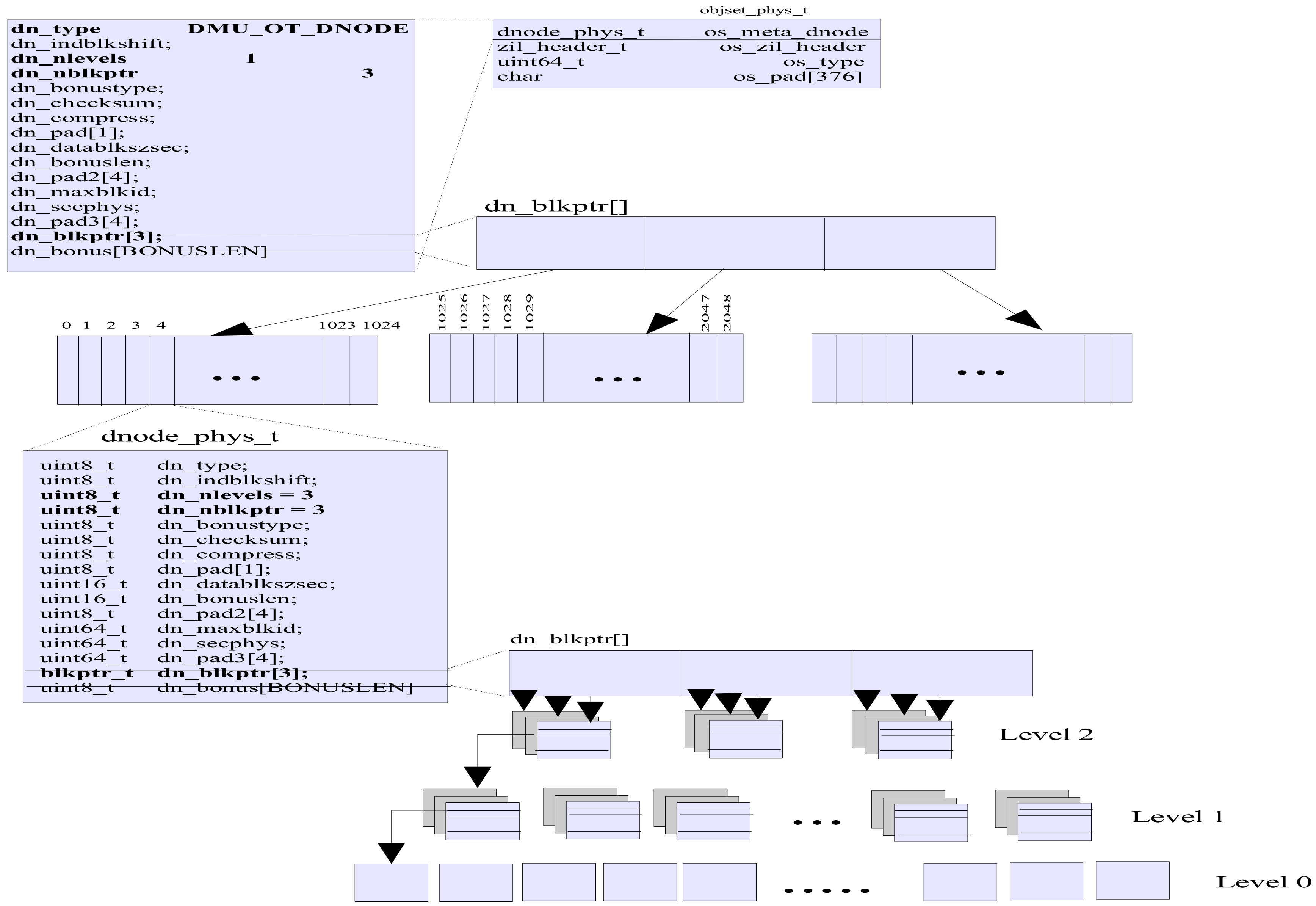
To achieve a maximum of number of blkptrs we steal space from the dn_bonus member

Use the DN_BONUS() macro to determine where the bonus buffer actually exists

```
#define DN_BONUS(dnp) ((void*)((dnp)->dn_bonus + \
        (((dnp)->dn_nblkptr - 1) *
sizeof(blkptr_t))))
```

In practice only one block pointer is used

Metadnode



dn_type	**DMU_OT_DNODE**
dn_indblkshift;	
dn_nlevels	**1**
dn_nblkptr	**3**
dn_bonustype;	
dn_checksum;	
dn_compress;	
dn_pad[1];	
dn_datablkszsec;	
dn_bonuslen;	
dn_pad2[4];	
dn_maxblkid;	
dn_secphys;	
dn_pad3[4];	
dn_blkptr[3];	
dn_bonus[BONUSLEN]	

objset_phys_t

dnode_phys_t	os_meta_dnode
zil_header_t	os_zil_header
uint64_t	os_type
char	os_pad[376]

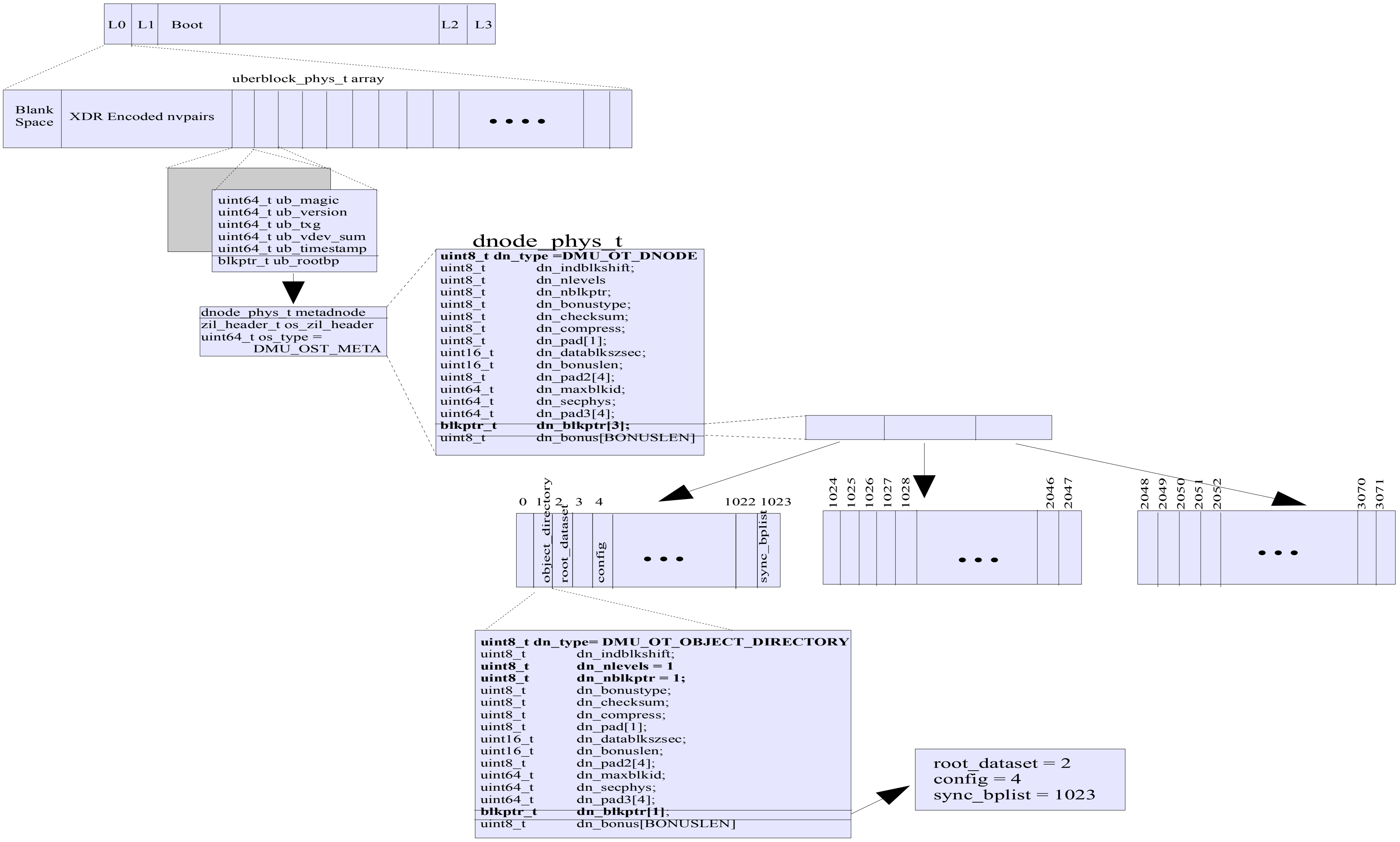
dn_blkptr[]

dnode_phys_t

uint8_t	dn_type;
uint8_t	dn_indblkshift;
uint8_t	**dn_nlevels = 3**
uint8_t	**dn_nblkptr = 3**
uint8_t	dn_bonustype;
uint8_t	dn_checksum;
uint8_t	dn_compress;
uint8_t	dn_pad[1];
uint16_t	dn_datablkszsec;
uint16_t	dn_bonuslen;
uint8_t	dn_pad2[4];
uint64_t	dn_maxblkid;
uint64_t	dn_secphys;
uint64_t	dn_pad3[4];
blkptr_t	**dn_blkptr[3];**
uint8_t	dn_bonus[BONUSLEN]

dn_blkptr[]

Complete View



L0 | L1 | Boot | L2 | L3

uberblock_phys_t array

Blank Space | XDR Encoded nvpairs | . . . .

```
uint64_t ub_magic
uint64_t ub_version
uint64_t ub_txg
uint64_t ub_vdev_sum
uint64_t ub_timestamp
blkptr_t ub_rootbp
```

```
dnode_phys_t metadnode
zil_header_t os_zil_header
uint64_t os_type =
        DMU_OST_META
```

dnode_phys_t

```
uint8_t dn_type =DMU_OT_DNODE
uint8_t          dn_indblkshift;
uint8_t          dn_nlevels;
uint8_t          dn_nblkptr;
uint8_t          dn_bonustype;
uint8_t          dn_checksum;
uint8_t          dn_compress;
uint8_t          dn_pad[1];
uint16_t         dn_datablkszsec;
uint16_t         dn_bonuslen;
uint8_t          dn_pad2[4];
uint64_t         dn_maxblkid;
uint64_t         dn_secphys;
uint64_t         dn_pad3[4];
blkptr_t         dn_blkptr[3];
uint8_t          dn_bonus[BONUSLEN]
```

0 1 2 3 4 1022 1023

object_directory | root_dataset | config | . . . | sync_bplist

1024 1025 1026 1027 1028 . . . 2046 2047

2048 2049 2050 2051 2052 . . . 3070 3071

```
uint8_t dn_type= DMU_OT_OBJECT_DIRECTORY
uint8_t          dn_indblkshift;
uint8_t          dn_nlevels = 1
uint8_t          dn_nblkptr = 1;
uint8_t          dn_bonustype;
uint8_t          dn_checksum;
uint8_t          dn_compress;
uint8_t          dn_pad[1];
uint16_t         dn_datablkszsec;
uint16_t         dn_bonuslen;
uint8_t          dn_pad2[4];
uint64_t         dn_maxblkid;
uint64_t         dn_secphys;
uint64_t         dn_pad3[4];
blkptr_t         dn_blkptr[1];
uint8_t          dn_bonus[BONUSLEN]
```

root_dataset = 2
config = 4
sync_bplist = 1023