ZFS

The Last Word In File Systems

Jeff Bonwick
Bill Moore
www.opensolaris.org/os/community/zfs

ZFS Overview

- Pooled storage
  - Completely eliminates the antique notion of volumes
  - Does for storage what VM did for memory
- Transactional object system
  - Always consistent on disk – no fsck, ever
  - Universal – file, block, iSCSI, swap ...
- Provable end-to-end data integrity
  - Detects and corrects silent data corruption
  - Historically considered "too expensive" – no longer true
- Simple administration
  - Concisely express your intent

Trouble with Existing Filesystems

- No defense against silent data corruption
  - Any defect in disk, controller, cable, driver, laser, or firmware can corrupt data silently; like running a server without ECC memory
- Brutal to manage
  - Labels, partitions, volumes, provisioning, grow/shrink, /etc files...
  - Lots of limits: filesystem/volume size, file size, number of files, files per directory, number of snapshots ...
  - Different tools to manage file, block, iSCSI, NFS, CIFS ...
  - Not portable between platforms (x86, SPARC, PowerPC, ARM ...)
- Dog slow
  - Linear-time create, fat locks, fixed block size, naïve prefetch, dirty region logging, painful RAID rebuilds, growing backup time

ZFS Objective

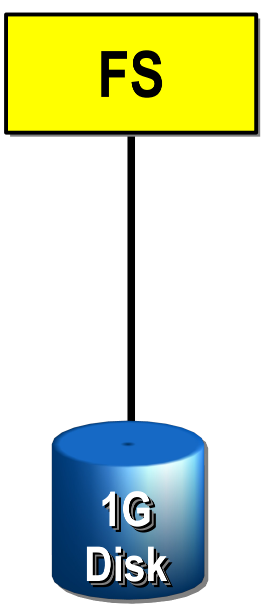
End the Suffering

- Figure out why storage has gotten so complicated
- Blow away 20 years of obsolete assumptions
- Design an integrated system from scratch

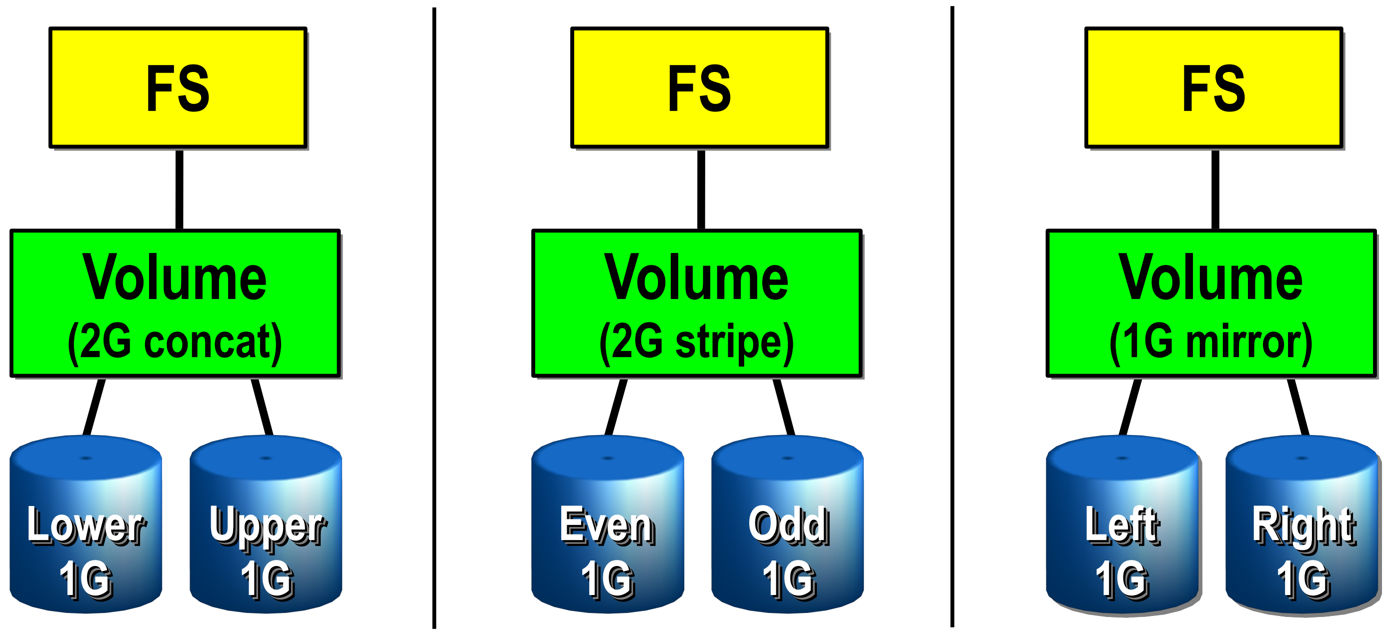
Why Volumes Exist

In the beginning, each filesystem managed a single disk.

It wasn't very big.



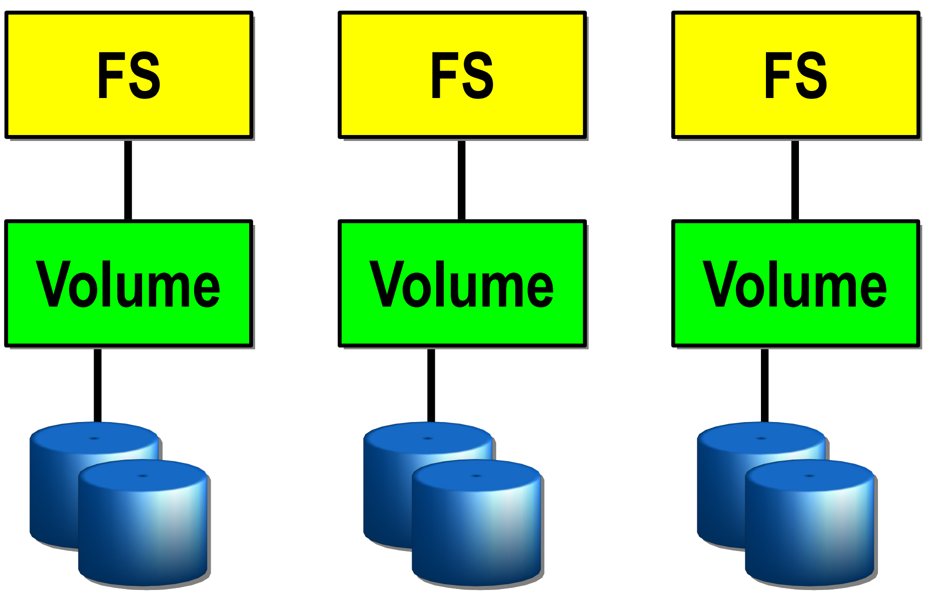
- Customers wanted more space, bandwidth, reliability
  - Hard: redesign filesystems to solve these problems well
  - Easy: insert a shim ("volume") to cobble disks together
- An industry grew up around the FS/volume model
  - Filesystem, volume manager sold as separate products
  - Inherent problems in FS/volume interface can't be fixed



FS/Volume Model vs. Pooled Storage

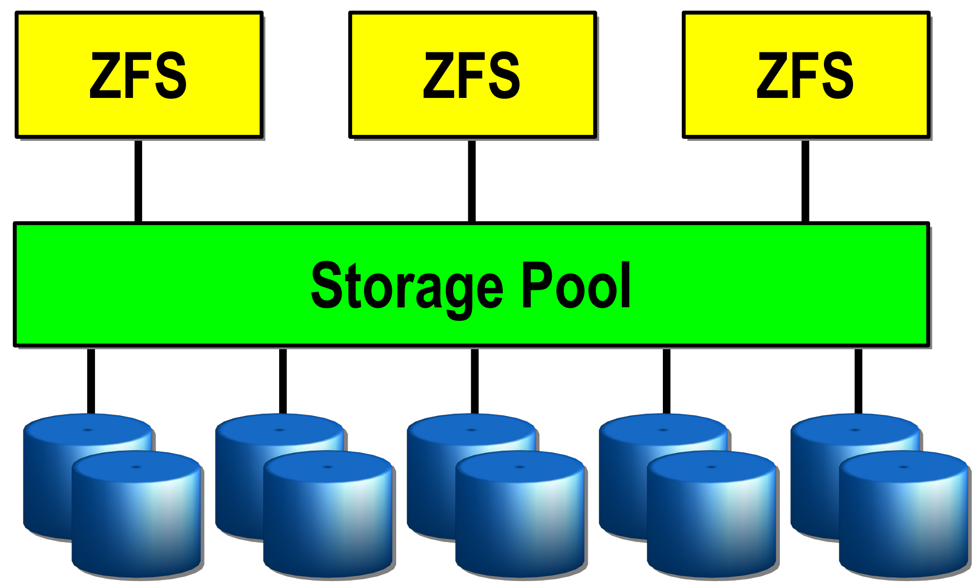
Traditional Volumes

- Abstraction: virtual disk
- Partition/volume for each FS
- Grow/shrink by hand
- Each FS has limited bandwidth
- Storage is fragmented, stranded



ZFS Pooled Storage

- Abstraction: malloc/free
- No partitions to manage
- Grow/shrink automatically
- All bandwidth always available
- All storage in the pool is shared



FS/Volume Interfaces vs. ZFS

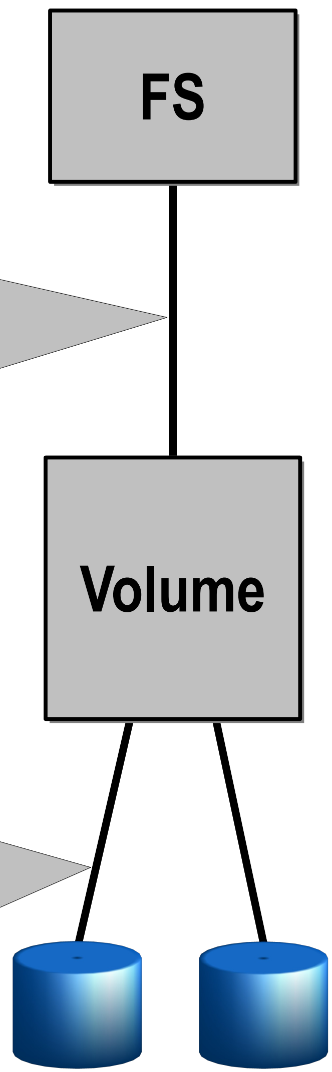
FS/Volume I/O Stack

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

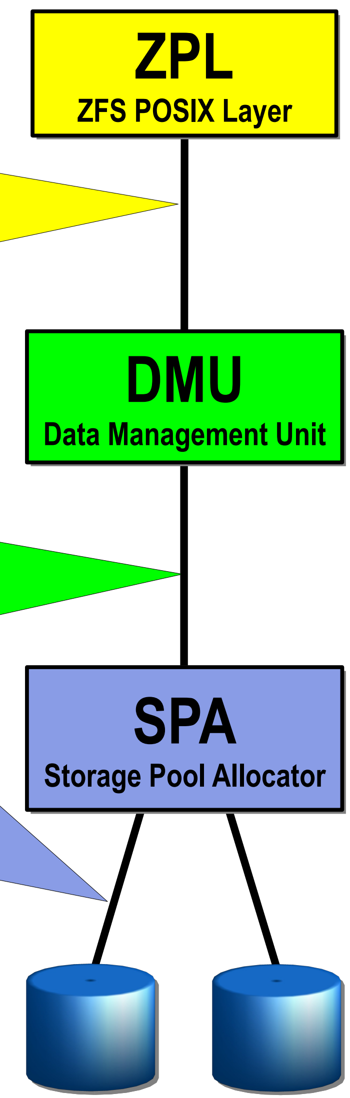
- "Make these 7 changes to these 3 objects"
- Atomic (all-or-nothing)

Transaction Group Commit

- Atomic for entire group
- Always consistent on disk
- No journal – not needed

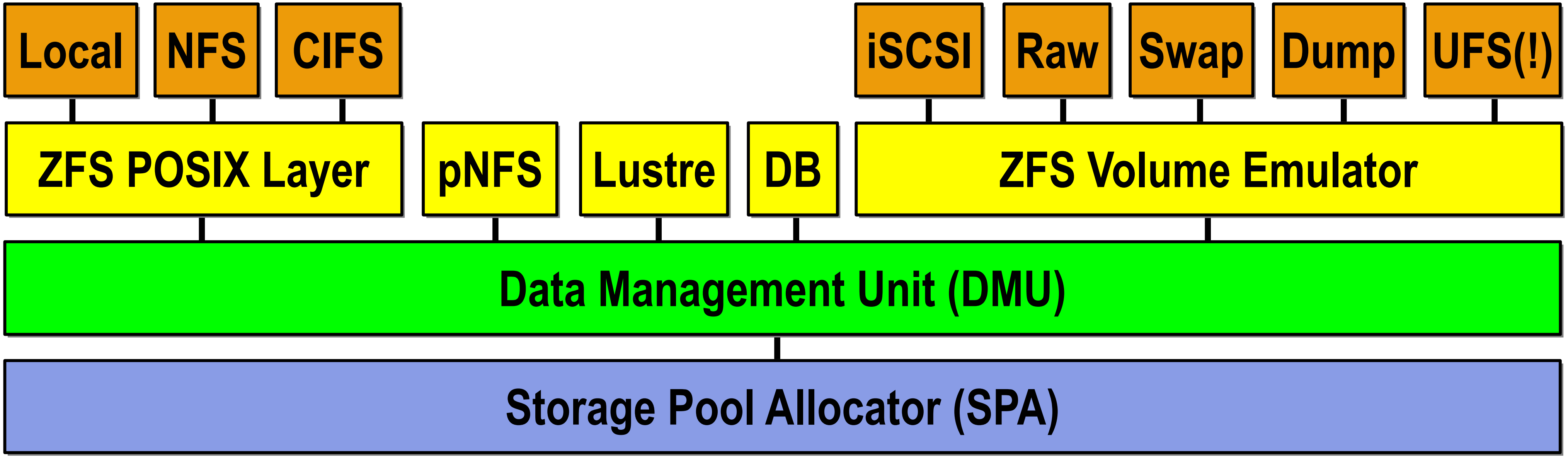
Transaction Group Batch I/O

- Schedule, aggregate, and issue I/O at will
- No resync if power lost
- Runs at platter speed



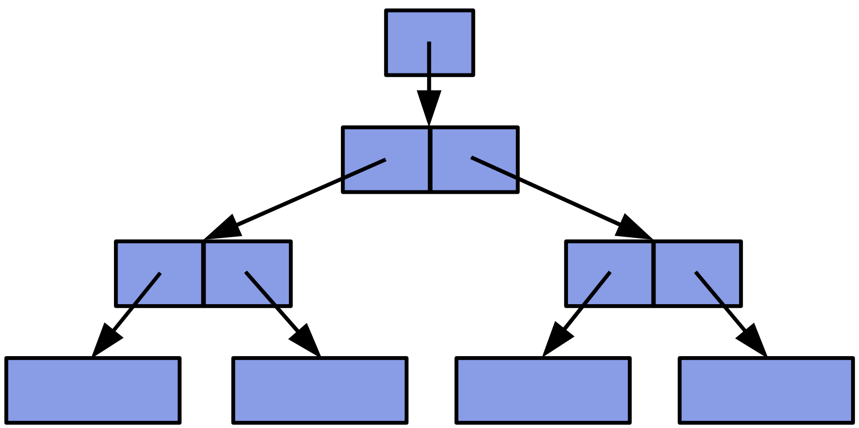
Universal Storage

- File, block, object – all in the same pool
  - DMU provides a general-purpose transactional object store
  - Pool = up to 2^{48} datasets, each up to 2^{48} objects, each up to 2^{64} bytes
- Key features are common to all datasets
  - Snapshots, compression, encryption, end-to-end data integrity

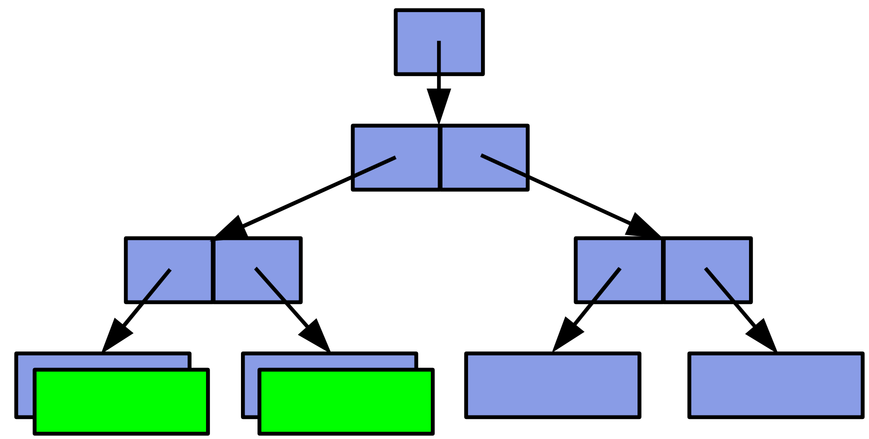


Copy-On-Write Transactions

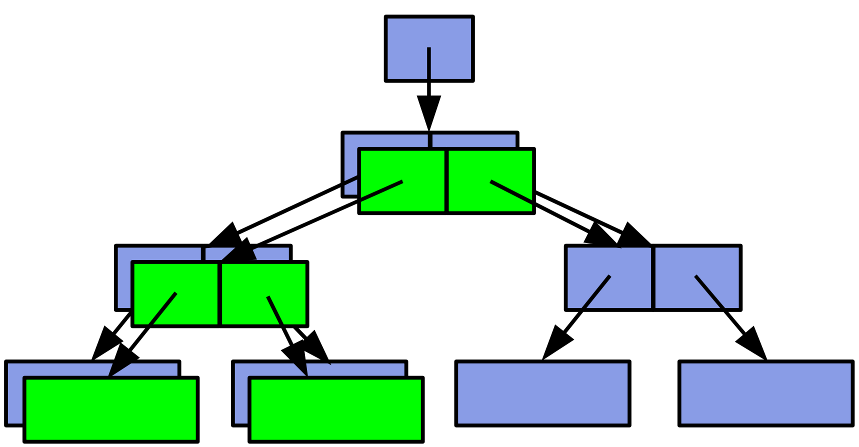
1. Initial block tree



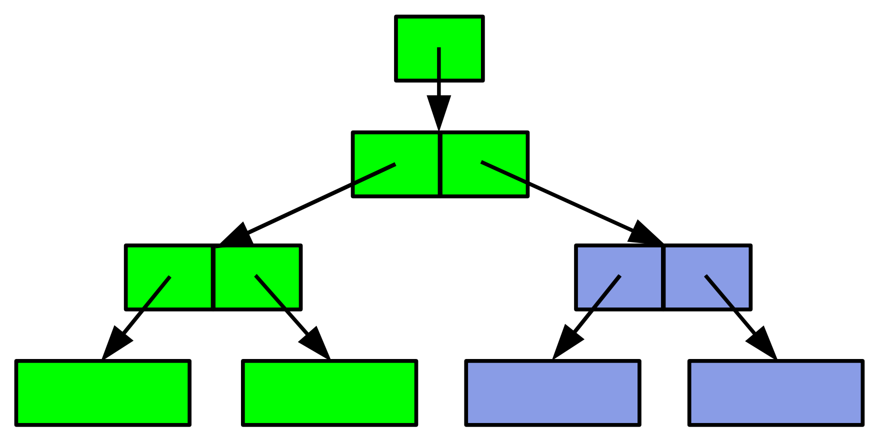
2. COW some blocks



3. COW indirect blocks

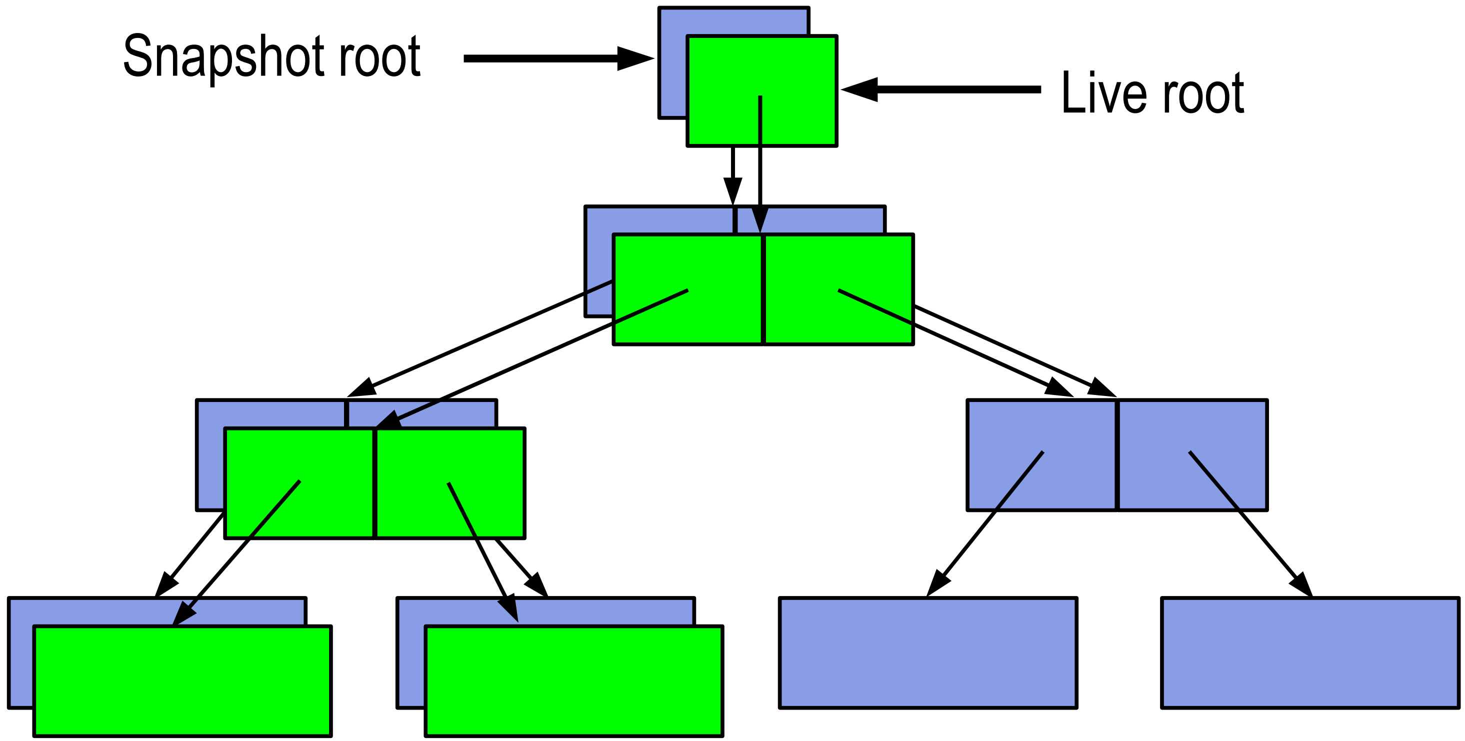


4. Rewrite uberblock (atomic)



Bonus: Constant-Time Snapshots

- The easy part: at end of TX group, don't free COWed blocks
  - Actually cheaper to take a snapshot than not!



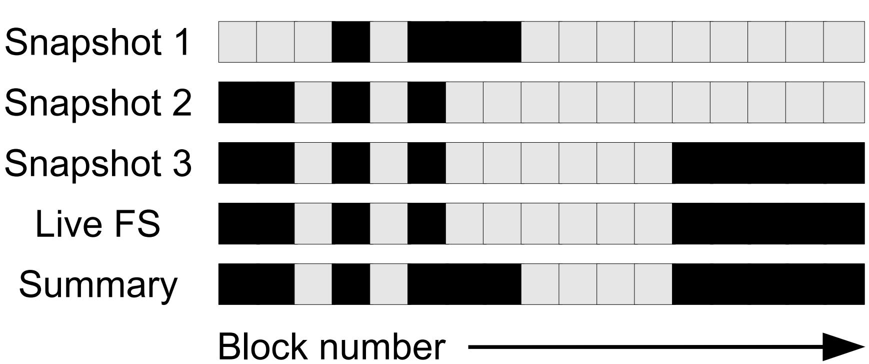
- The tricky part: how do you know when a block is free?

Traditional Snapshots vs. ZFS

STORAGE DEVELOPER CONFERENCE
SNIA SANTA CLARA, 2009

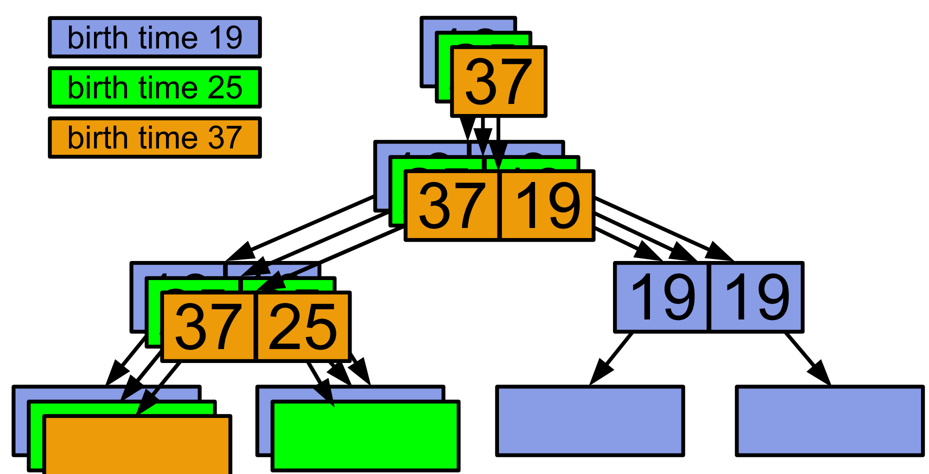
Per-Snapshot Bitmaps

- Block allocation bitmap for every snapshot
  - O(N) per-snapshot space overhead
  - Limits number of snapshots
- O(N) create, O(N) delete, O(N) incremental
  - Snapshot bitmap comparison is O(N)
  - Generates unstructured block delta
  - Requires some prior snapshot to exist



ZFS Birth Times

- Each block pointer contains child's birth time
  - O(1) per-snapshot space overhead
  - Unlimited snapshots
- O(1) create, O(Δ) delete, O(Δ) incremental
  - Birth-time-pruned tree walk is O(Δ)
  - Generates semantically rich object delta
  - Can generate delta since any point in time



Trends in Storage Integrity

- Uncorrectable bit error rates have stayed roughly constant
    - 1 in 10^{14} bits (~12TB) for desktop-class drives
    - 1 in 10^{15} bits (~120TB) for enterprise-class drives (allegedly)
    - Bad sector every 8-20TB in practice (desktop <u>and</u> enterprise)
- Drive capacities doubling every 12-18 months
- Number of drives per deployment increasing
- ⟶ Rapid increase in error rates
- Both silent and "noisy" data corruption becoming more common
- Cheap flash storage will only accelerate this trend

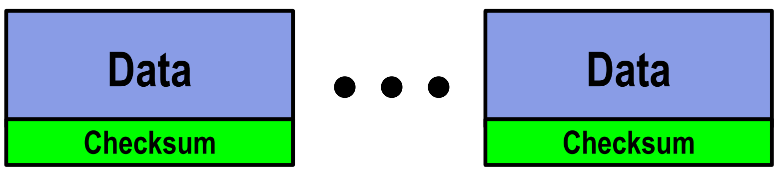
Measurements at CERN

- Wrote a simple application to write/verify 1GB file
  - Write 1MB, sleep 1 second, etc. until 1GB has been written
  - Read 1MB, verify, sleep 1 second, etc.
- Ran on 3000 rack servers with HW RAID card
- After 3 weeks, found 152 instances of silent data corruption
  - Previously thought "everything was fine"
- HW RAID only detected "noisy" data errors
- Need end-to-end verification to catch silent data corruption

End-to-End Data Integrity in ZFS

Disk Block Checksums

- Checksum stored with data block
- Any self-consistent block will pass
- Can't detect stray writes
- Inherent FS/volume interface limitation

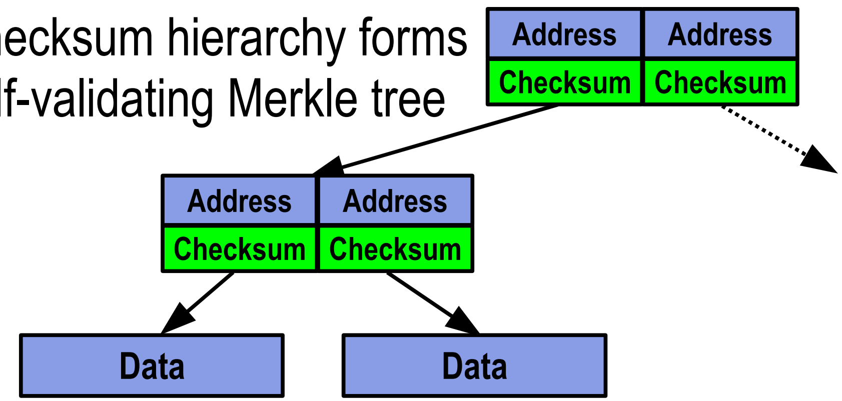


Disk checksum only validates media

- ✔ 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
- Checksum hierarchy forms self-validating Merkle tree

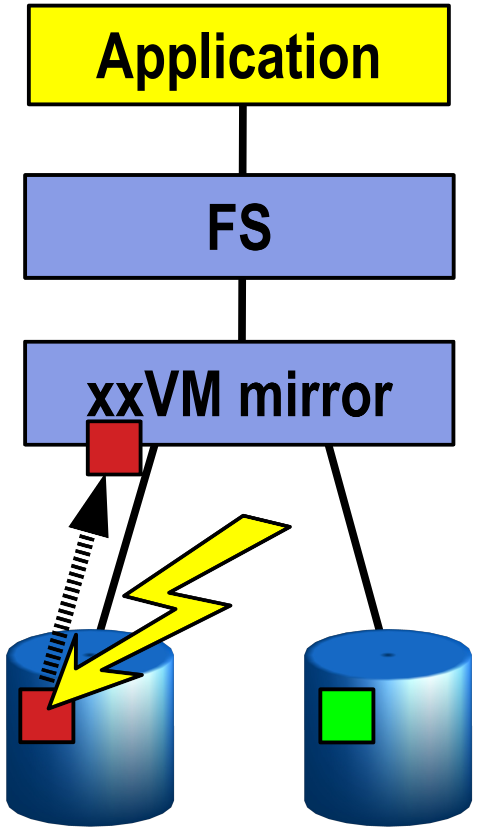


ZFS validates the entire I/O path

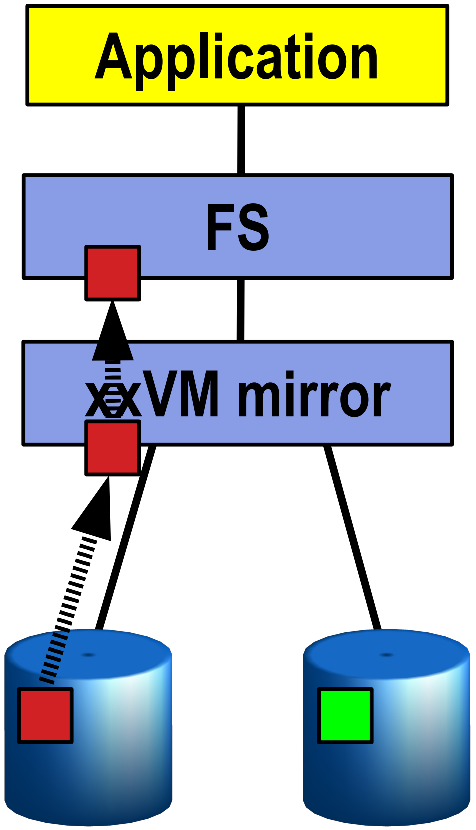
- ✔ Bit rot
- ✔ Phantom writes
- ✔ Misdirected reads and writes
- ✔ DMA parity errors
- ✔ Driver bugs
- ✔ Accidental overwrite

Traditional Mirroring

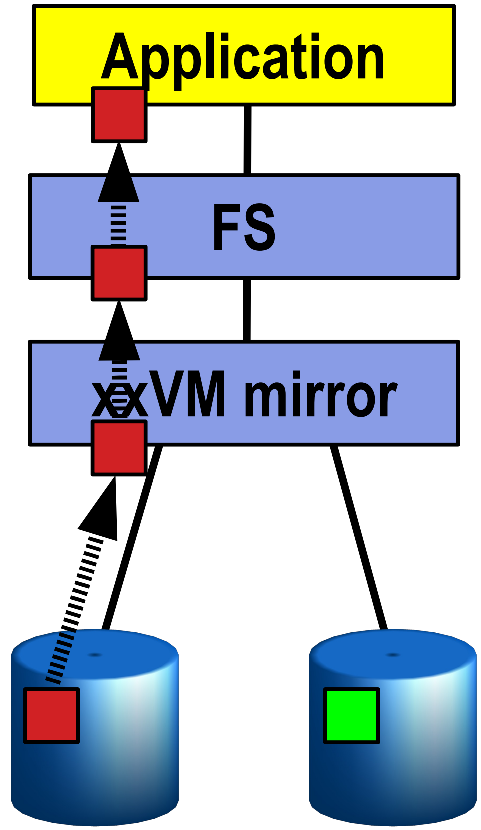
1. Application issues a read. Mirror reads the first disk, which has a corrupt block. It can't tell.



2. Volume manager passes bad block up to filesystem. If it's a metadata block, the filesystem panics. If not...



3. Filesystem returns bad data to the application.

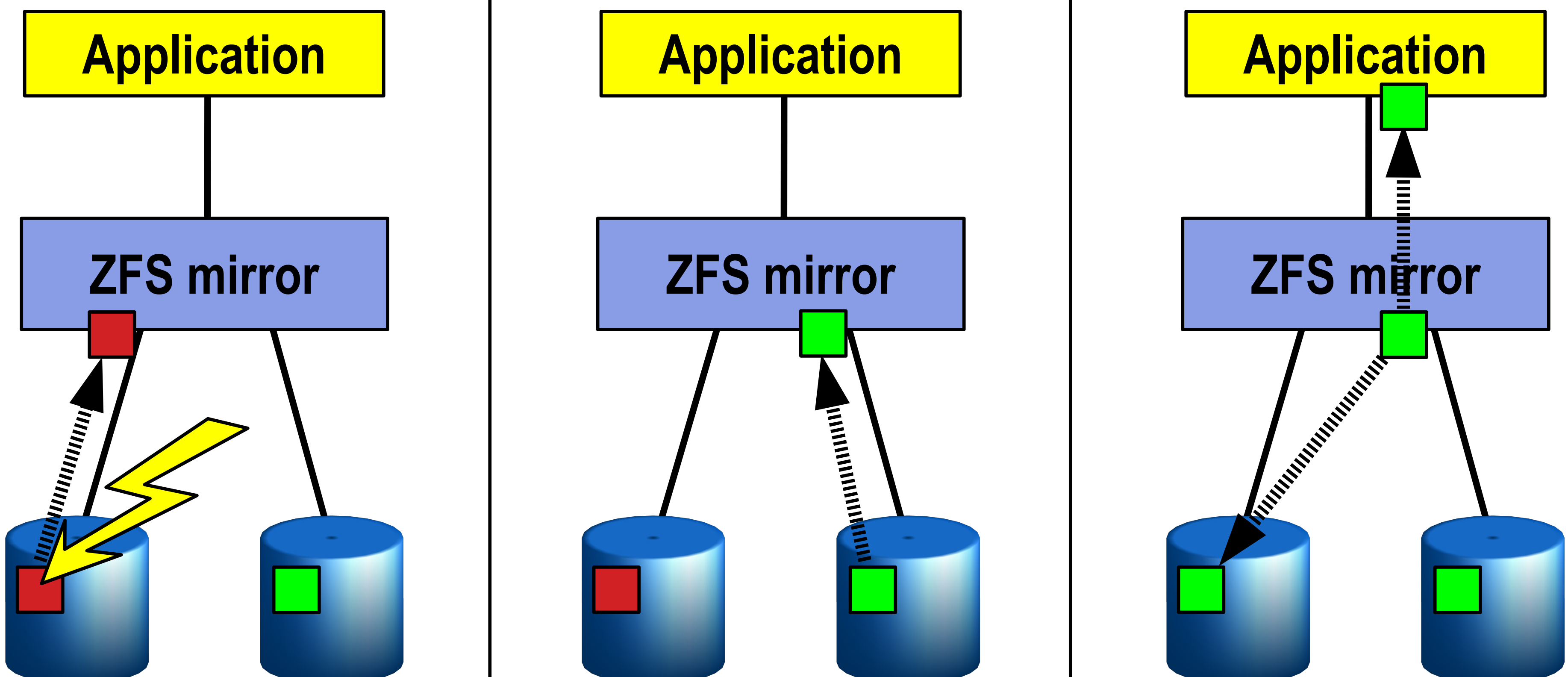


Self-Healing Data in ZFS

1. Application issues a read. ZFS mirror tries the first disk. Checksum reveals that the block is corrupt on disk.

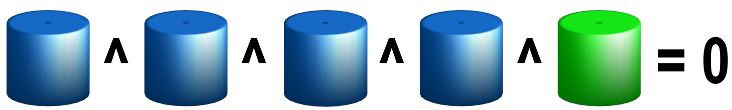
2. ZFS tries the second disk. Checksum indicates that the block is good.

3. ZFS returns known good data to the application and repairs the damaged block.

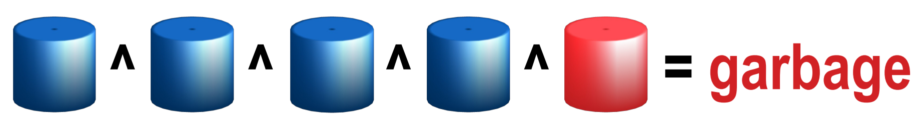


Traditional RAID-4 and RAID-5

- Several data disks plus one parity disk



- Fatal flaw: partial stripe writes
  - Parity update requires read-modify-write (slow)
    - Read old data and old parity (two synchronous disk reads)
    - Compute new parity = new data ^ old data ^ old parity
    - Write new data and new parity
  - Suffers from *write hole:*



    - Loss of power between data and parity writes will corrupt data
    - Workaround: $$$ NVRAM in hardware (i.e., don't lose power!)
- Can't detect or correct silent data corruption

RAID-Z

- Dynamic stripe width
  - Variable block size: 512 – 128K
  - Each logical block is its own stripe
- All writes are full-stripe writes
  - Eliminates read-modify-write (it's fast)
  - Eliminates the RAID-5 write hole (no need for NVRAM)
- Single, double, and triple parity
- Detects and corrects silent data corruption
  - Checksum-driven combinatorial reconstruction
- No special hardware – ZFS loves cheap disks

LBA \ Disk	A	B	C	D	E
0	P_0	D_0	D_2	D_4	D_6
1	P_1	D_1	D_3	D_5	D_7
2	P_0	D_0	D_1	D_2	P_0
3	D_0	D_1	D_2	P_0	D_0
4	P_0	D_0	D_4	D_8	D_{11}
5	P_1	D_1	D_5	D_9	D_{12}
6	P_2	D_2	D_6	D_{10}	D_{13}
7	P_3	D_3	D_7	P_0	D_0
8	D_1	D_2	D_3	X	P_0
9	D_0	D_1	X	P_0	D_0
10	D_3	D_6	D_9	P_1	D_1
11	D_4	D_7	D_{10}	P_2	D_2
12	D_5	D_8	•	•	•

Traditional Resilvering

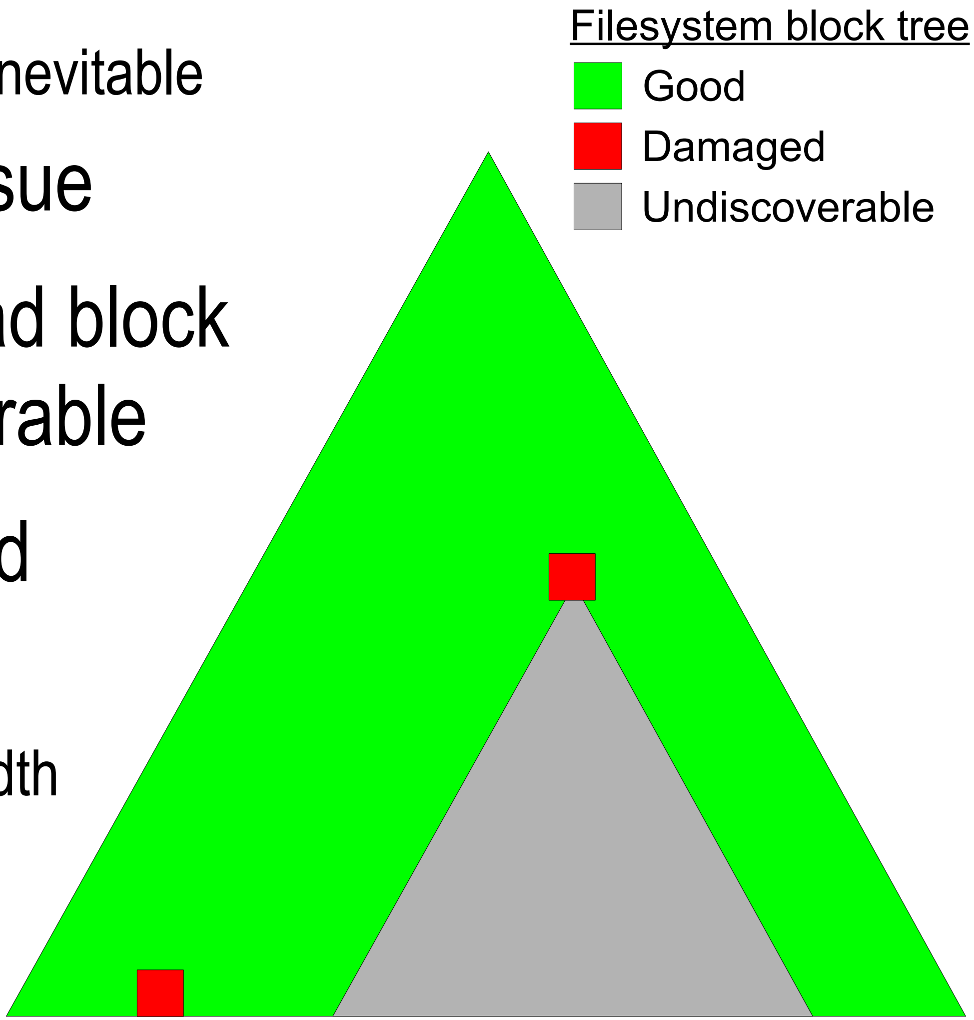
- Creating a new mirror (or RAID stripe):
  - Copy one disk to the other (or XOR them together) so all copies are self-consistent – even though they're all random garbage!
- Replacing a failed device:
  - Whole-disk copy – even if the volume is nearly empty
  - No checksums or validity checks along the way
  - No assurance of progress until 100% complete – your root directory may be the last block copied
- Recovering from a transient outage:
  - Dirty region logging – slow, and easily defeated by random writes

Smokin' Mirrors

- Top-down resilvering
  - ZFS resilvers the storage pool's block tree from the root down
  - Most important blocks first
  - Every single block copy increases the amount of discoverable data
- Only copy live blocks
  - No time wasted copying free space
  - Zero time to initialize a new mirror or RAID-Z group
- Dirty time logging (for transient outages)
  - ZFS records the transaction group window that the device missed
  - To resilver, ZFS walks the tree and prunes where birth time < DTL
  - A five-second outage takes five seconds to repair

Surviving Multiple Data Failures

- With increasing error rates, multiple failures can exceed RAID's ability to recover
  - With a big enough data set, it's inevitable
- Silent errors compound the issue
- Filesystems are trees: one bad block can render millions undiscoverable
- "More important" blocks should be more highly replicated
  - Small cost in space and bandwidth
  - Huge gain in robustness



Ditto Blocks

- Data replication above and beyond mirror/RAID-Z
    - Each logical block can have up to three physical blocks
        - Different devices whenever possible
        - Different places on the same device otherwise (e.g. laptop drive)
    - All ZFS metadata 2+ copies
        - Small cost in latency and bandwidth (metadata ≈ 1% of data)
    - Explicitly settable for precious user data
- Detects and corrects silent data corruption
    - In a multi-disk pool, ZFS survives any non-consecutive disk failures
    - In a single-disk pool, ZFS survives loss of up to 1/8 of the platter
- ZFS survives failures that send other filesystems to tape

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
        - All mirror copies, all RAID-Z parity, and all ditto blocks
    - Verifies each copy against its 256-bit checksum
    - Repairs data as it goes
- Minimally invasive
    - Low I/O priority ensures that scrubbing doesn't get in the way
    - User-defined scrub rates coming soon
        - Gradually scrub the pool over the course of a month, a quarter, etc.

ZFS Scalability

- Immense capacity (128-bit)
  - Moore's Law: need 65th bit in 10-15 years
  - ZFS capacity: 256 quadrillion ZB (1ZB = 1 billion TB)
  - Exceeds quantum limit of Earth-based storage
    - Seth Lloyd, "Ultimate physical limits to computation." Nature 406, 1047-1054 (2000)
- 100% dynamic metadata
  - No limits on files, directory entries, etc.
  - No wacky knobs (e.g. inodes/cg)
- Concurrent everything
  - Byte-range locking: parallel read/write without violating POSIX
  - Parallel, constant-time directory operations

ZFS Performance

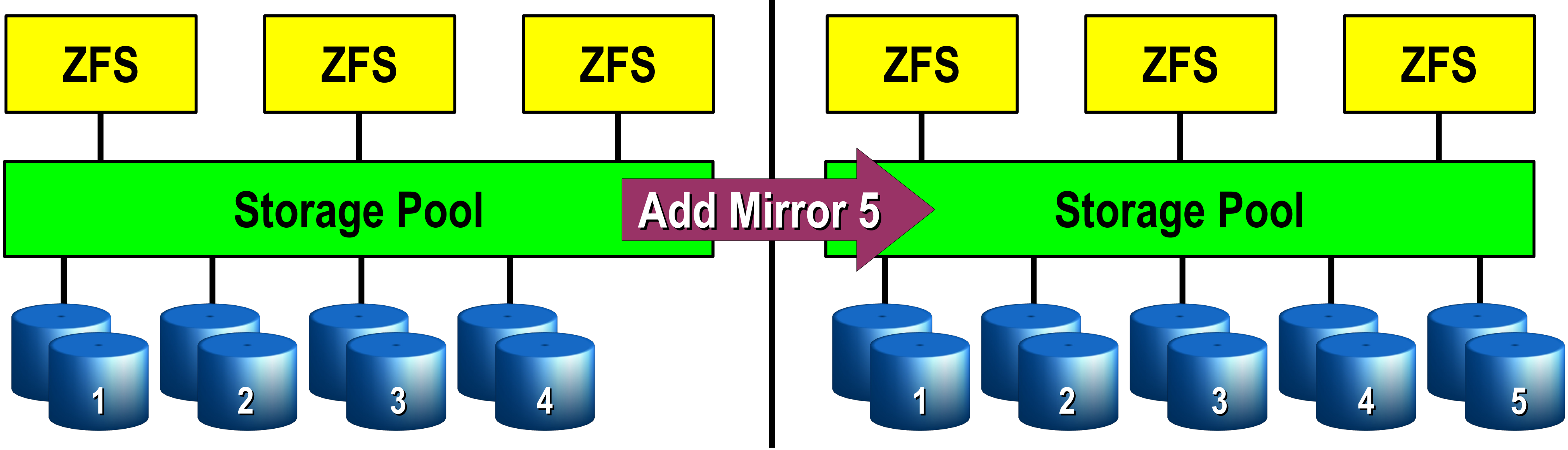
- Copy-on-write design
  - Turns random writes into sequential writes
  - Intrinsically hot-spot-free
- Pipelined I/O
  - Fully scoreboarded 24-stage pipeline with I/O dependency graphs
  - Maximum possible I/O parallelism
  - Priority, deadline scheduling, out-of-order issue, sorting, aggregation
- Dynamic striping across all devices
- Intelligent prefetch
- Variable block size
- Integrated flash memory support

Dynamic Striping

- Automatically distributes load across all devices

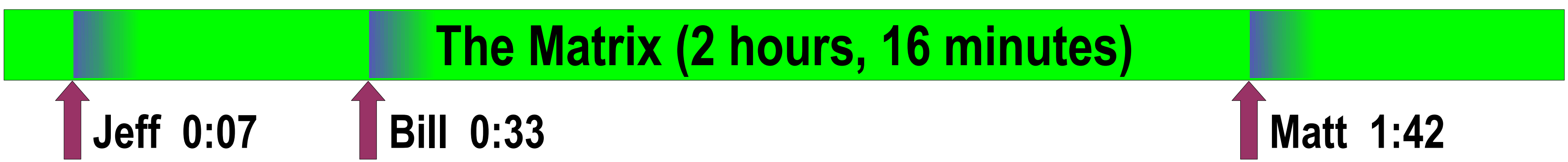
- Writes: striped across all four mirrors
- Reads: wherever the data was written
- Block allocation policy considers:
  - Capacity
  - Performance (latency, BW)
  - Health (degraded mirrors)

- 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

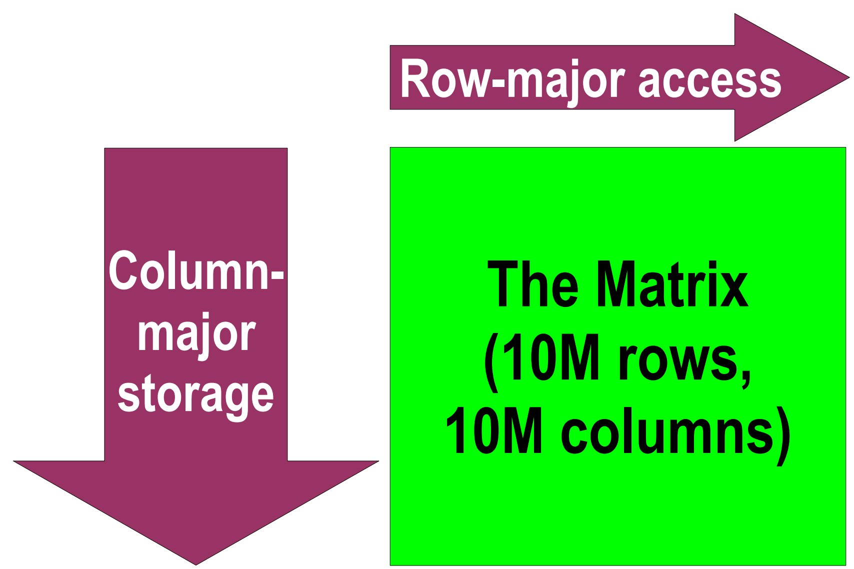


Intelligent Prefetch

- Multiple independent prefetch streams
  - Crucial for any streaming service provider



- Automatic length and stride detection
  - Great for HPC applications
  - ZFS understands the matrix multiply problem
    - Detects any linear access pattern
    - Forward or backward

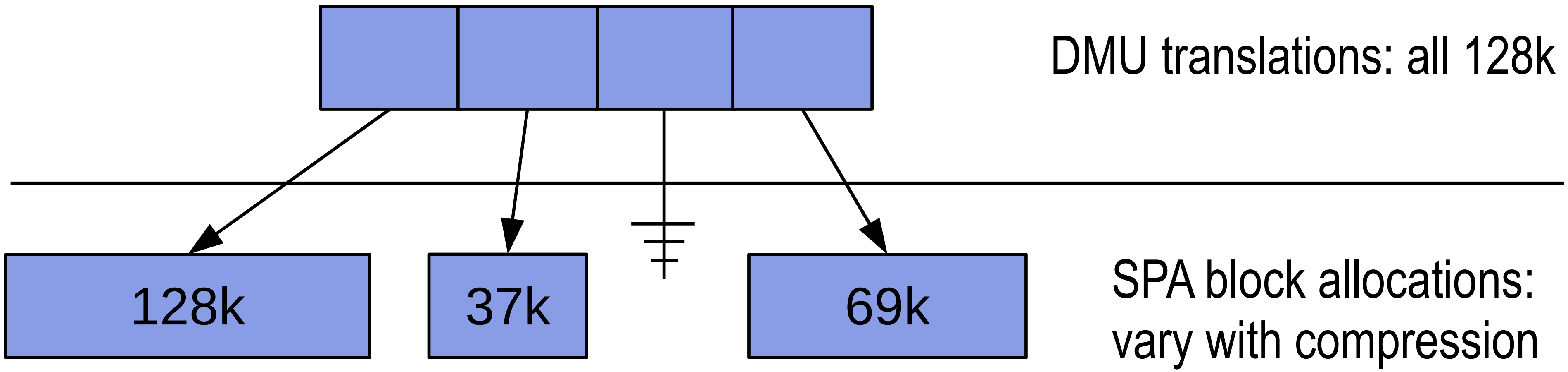


Variable Block Size

- No single block size is optimal for everything
  - Large blocks: less metadata, higher bandwidth
  - Small blocks: more space-efficient for small objects
  - Record-structured files (e.g. databases) have natural granularity; filesystem must match it to avoid read/modify/write
- Why not arbitrary extents?
  - Extents don't COW or checksum nicely (too big)
  - Large blocks suffice to run disks at platter speed
- Per-object granularity
  - A 37k file consumes 37k – no wasted space
- Enables transparent block-based compression

Built-in Compression

- Block-level compression in SPA
  - Transparent to all other layers
  - Each block compressed independently
  - All-zero blocks converted into file holes



- LZJB and GZIP available today; more on the way

Integrated Flash Memory Support

- Write side:
  - Dedicated log devices for fast synchronous writes
    - A few GB is plenty
    - Large and/or asynchronous writes go straight to disk
  - Ideal device: enterprise-grade SLC flash SSD
    - Much cheaper than NVRAM
    - Easily clustered over standard SAS fabric
- Read side:
  - Dedicated cache devices for fast random reads
    - An eviction cache for DRAM
    - As much as necessary to hold the working set
  - Ideal device: big, cheap, consumer-grade MLC flash
    - It's just a cache – failures are OK
    - Everything is checksummed – no risk of silent errors

ZFS Hybrid Pool Example

- 4 Xeon 7350 Processors (16 cores)
- 32GB FB DDR2 ECC DRAM
- OpenSolaris with ZFS

Configuration A:

(7) 146GB 10,000 RPM SAS Drives

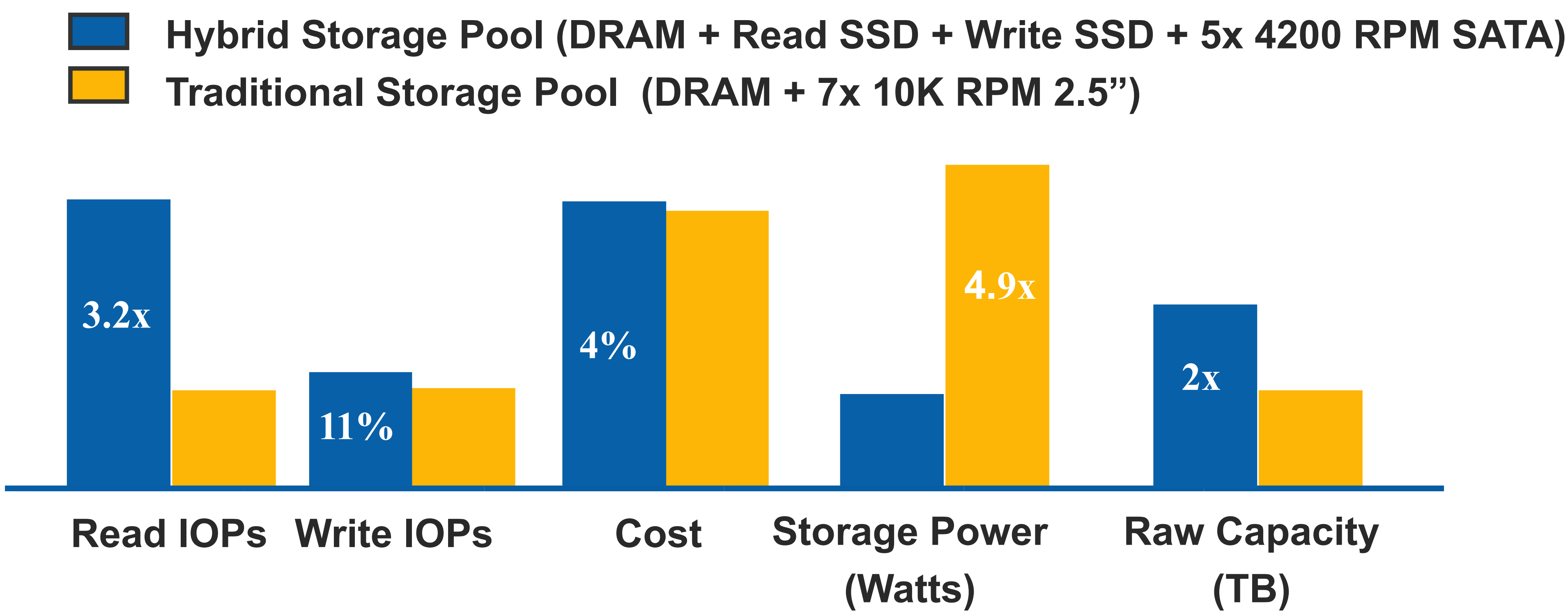
Configuration B:

(1) 32G SSD Log Device

(1) 80G SSD Cache Device

(5) 400GB 4200 RPM SATA Drives

ZFS Hybrid Pool Performance



- If NVRAM were used, hybrid wins on cost, too
- For large configs (50T - 1PB+) cost is entirely amortized

ZFS Administration

- Pooled storage – no more volumes!
  - Up to 2^{48} datasets per pool – filesystems, iSCSI targets, swap, etc.
  - Nothing to provision!
- Filesystems become administrative control points
  - Hierarchical, with inherited properties
    - Per-dataset policy: snapshots, compression, backups, quotas, etc.
    - Who's using all the space? du(1) takes forever, but df(1M) is instant
    - Manage logically related filesystems as a group
    - Inheritance makes large-scale administration a snap
  - Policy follows the data (mounts, shares, properties, etc.)
  - Delegated administration lets users manage their own data
  - ZFS filesystems are cheap – use a ton, it's OK, really!
- Online everything

Creating Pools and Filesystems

- Create a mirrored pool named "tank"

```
# zpool create tank mirror c2d0 c3d0
```

- Create home directory filesystem, mounted at /export/home

```
# zfs create tank/home
# zfs set mountpoint=/export/home tank/home
```

- Create home directories for several users
  Note: automatically mounted at /export/home/{ahrens,bonwick,billm} thanks to inheritance

```
# zfs create tank/home/ahrens
# zfs create tank/home/bonwick
# zfs create tank/home/billm
```

- Add more space to the pool

```
# zpool add tank mirror c4d0 c5d0
```

Setting Properties

- Automatically NFS-export all home directories

```
# zfs set sharenfs=rw tank/home
```

- Turn on compression for everything in the pool

```
# zfs set compression=on tank
```

- Limit Eric to a quota of 10g

```
# zfs set quota=10g tank/home/eschrock
```

- Guarantee Tabriz a reservation of 20g

```
# zfs set reservation=20g tank/home/tabriz
```

ZFS Snapshots

- Read-only point-in-time copy of a filesystem
  - Instantaneous creation, unlimited number
  - No additional space used – blocks copied only when they change
  - Accessible through .zfs/snapshot in root of each filesystem
    - Allows users to recover files without sysadmin intervention
- Take a snapshot of Mark's home directory

```
# zfs snapshot tank/home/marks@tuesday
```

- Roll back to a previous snapshot

```
# zfs rollback tank/home/perrin@monday
```

- Take a look at Wednesday's version of foo.c

```
$ cat ~maybee/.zfs/snapshot/wednesday/foo.c
```

ZFS Clones

- Writable copy of a snapshot
  - Instantaneous creation, unlimited number
- Ideal for storing many private copies of mostly-shared data
  - Software installations
  - Source code repositories
  - Diskless clients
  - Zones
  - Virtual machines

- Create a clone of your OpenSolaris source code

```
# zfs clone tank/solaris@monday tank/ws/lori/fix
```

ZFS Send / Receive

- Powered by snapshots
  - Full backup: any snapshot
  - Incremental backup: any snapshot delta
  - Very fast delta generation – cost proportional to data changed
- So efficient it can drive remote replication
- Generate a full backup

```
# zfs send tank/fs@A >/backup/A
```

- Generate an incremental backup

```
# zfs send -i tank/fs@A tank/fs@B >/backup/B-A
```

- Remote replication: send incremental once per minute

```
# zfs send -i tank/fs@11:31 tank/fs@11:32 |
  ssh host zfs receive -d /tank/fs
```

ZFS Data Migration

- Host-neutral on-disk format
  - Change server from x86 to SPARC, it just works
  - Adaptive endianness: neither platform pays a tax
    - Writes always use native endianness, set bit in block pointer
    - Reads byteswap only if host endianness != block endianness
- ZFS takes care of everything
  - Forget about device paths, config files, /etc/vfstab, etc.
  - ZFS will share/unshare, mount/unmount, etc. as necessary
- Export pool from the old server

```
old# zpool export tank
```

- Physically move disks and import pool to the new server

```
new# zpool import tank
```

Native CIFS (SMB) Support

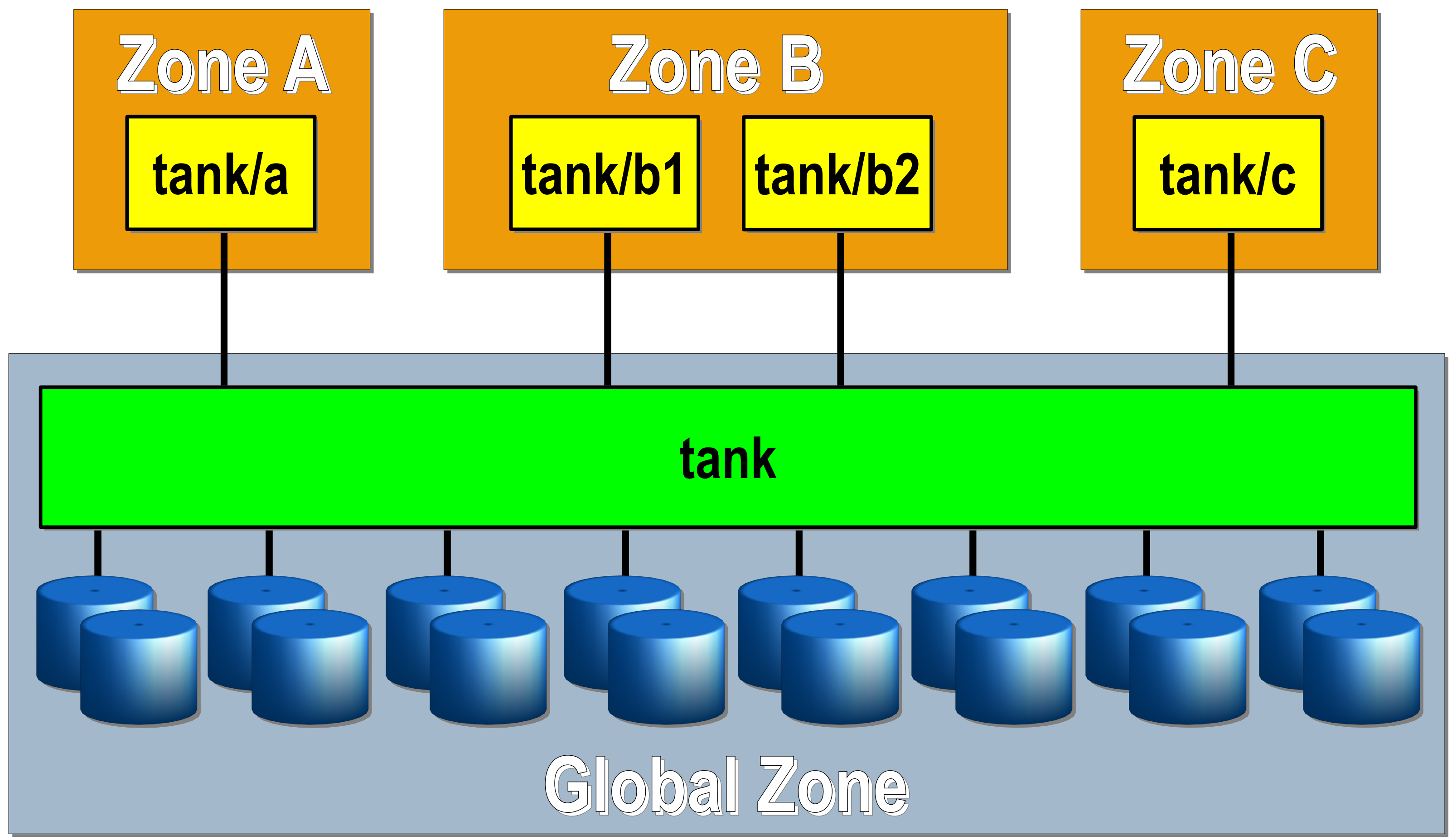
- NT-style ACLs
    - Allow/deny with inheritance
- True Windows SIDs – not just POSIX UID mapping
    - Essential for proper Windows interaction
    - Simplifies domain consolidation
- Options to control:
    - Case-insensitivity
    - Non-blocking mandatory locks
    - Unicode normalization
    - Virus scanning
- Simultaneous NFS and CIFS client access

ZFS and Zones (Virtualization)

- Secure – Local zones cannot even <u>see</u> physical devices
- Fast – snapshots and clones make zone creation instant

Dataset: Logical resource in local zone

Pool: Physical resource in global zone



ZFS Root

- Brings all the ZFS goodness to /
  - Checksums, compression, replication, snapshots, clones
  - Boot from any dataset
- Patching is always safe
  - Take snapshot, apply patch... rollback if there's a problem
- Live upgrade is incredibly fast
  - Create clone (instant), upgrade, boot from clone
- Supports multiple boot environments
  - Safe and easy to run experimental code
  - No partitions to manage

ZFS Test Methodology

- A product is only as good as its test suite
  - ZFS was designed to run in either user or kernel context
  - Nightly "ztest" program does all of the following in parallel:
    - Read, write, create, and delete files and directories
    - Create and destroy entire filesystems and storage pools
    - Turn compression on and off (while filesystem is active)
    - Change checksum algorithm (while filesystem is active)
    - Add and remove devices (while pool is active)
    - Change I/O caching and scheduling policies (while pool is active)
    - Scribble random garbage on one side of live mirror to test self-healing data
    - Force violent crashes to simulate power loss, then verify pool integrity
  - Probably more abuse in 20 seconds than you'd see in a lifetime
  - ZFS has been subjected to over a million forced, violent crashes without losing data integrity or leaking a single block

ZFS Summary

End the Suffering ● Free Your Mind

- Simple
  - Concisely expresses the user's intent
- Powerful
  - Pooled storage, snapshots, clones, compression, scrubbing, RAID-Z
- Safe
  - Detects and corrects silent data corruption
- Fast
  - Dynamic striping, intelligent prefetch, pipelined I/O
- Open
  - http://www.opensolaris.org/os/community/zfs
- Free

Where to Learn More

- Community: http://www.opensolaris.org/os/community/zfs
- Wikipedia: http://en.wikipedia.org/wiki/ZFS
- ZFS blogs: http://blogs.sun.com/main/tags/zfs
  - ZFS internals (snapshots, RAID-Z, dynamic striping, etc.)
  - Using iSCSI, CIFS, Zones, databases, remote replication and more
  - Latest news on pNFS, Lustre, and ZFS crypto projects
- ZFS ports
  - Apple Mac: http://developer.apple.com/adcnews
  - FreeBSD: http://wiki.freebsd.org/ZFS
  - Linux/FUSE: http://zfs-on-fuse.blogspot.com
  - As an appliance: http://www.nexenta.com

ZFS

The Last Word In File Systems

Jeff Bonwick
Bill Moore

www.opensolaris.org/os/community/zfs