CSE 451: Operating Systems

Spring 2022

Module 15.0

ZFS

The Zettabyte File System

John Zahorjan

This Module

- We have some slides I’ve built to give an overview
- We also have more details in:
  - an early overview paper
  - a somewhat more recent, commercial Powerpoint presentation
- Both are linked from the course calendar

Background

- We've looked at
  - FAT/NTFS/FFS – how to represent file system directory tree on disk; how to choose which blocks to allocate for metata and for file data
  - journaling – how to make the file system resilient to crashes
  - log structured FS – how to make all writes big, sequential writes
  - RAID – how to take advantage of "bytes are cheap" to obtain better performance, and how to deal with the elevated disk failure rate that comes from using more disks
  - *Backup – surviving user mistakes*
  - *Disk partitions – turning one physical device into multiple logical devices*

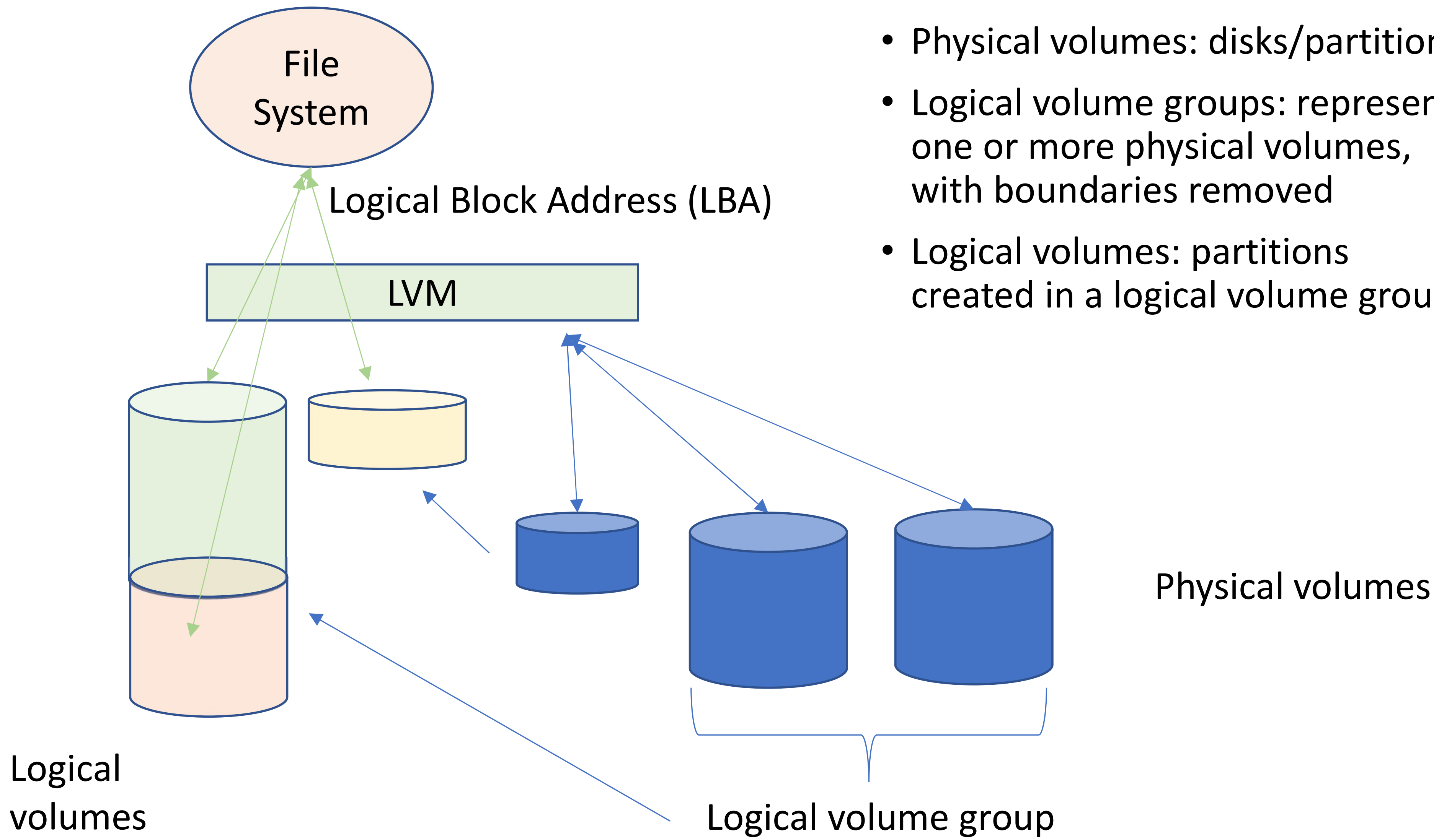
- ZFS comes later (around 2003)
- It is motivated by the difficulty of administering a system, especially one that has many disks and whose storage capacity may be changing
- *Deprecate taking the device from one system to another and having it be self-describing*
  - *ZFS provides tools to move the file system…*

ZFS

- Suppose you have a system with a single disk and it starts to fill. What do you do?
  - Buy a new disk twice as big, install the OS and apps on it, then copy the user files from the old disk to the new one?
  - Buy another disk the same size, keep it as is, and mount the new disk somewhere handy in the existing file system name space?
    - Do that but move some existing data files to the new disk?
  - What happens when the I run out of space again?

- One point of ZFS is that the boundaries of physical disks aren't sufficiently hidden by existing file systems

Logical Volume Managers (LVMs)



- Physical volumes: disks/partitions
- Logical volume groups: represent one or more physical volumes, with boundaries removed
- Logical volumes: partitions created in a logical volume group

LVMs

- LVMs can be in hardware (disk controllers) or software
- They can implement various RAID levels
- They can implement JBOD (Just a Bunch of Disks)
  - Aggregate storage blocks from many physical devices into one logical volume
  - No added error resilience

- RAIDs typically require many disks of the same capacity (and maybe type)
- JBOD doesn’t care what size they are

LVMs

- Okay, that's appealing for dealing with physical device boundaries
- Suppose you have formatted the logical volume for some file system (so superblock, free inode map, and inode arrays have been initialized and then used)
- Now you want to add storage to the system and then make the logical volume bigger
  - Can that work? Will the file system data structures on the logical volume be able to use the additional disk?
- Now you want to move space between one logical volume and another.
  - Can that work? Can you shrink a volume that holds files?

- One goal of ZFS is to address the difficult interplay among the physical devices, the logical devices, and the file systems

Error Resilience

- The only errors we have looked at are:
  - system crashes: journaling
  - disk dies: redundancy (RAID)

- What about:
  - disk has an undetected read error (returns incorrect data)?
  - disk has an undetected write error?
  - disk writes wrong block (controller or disk error)?
  - disk reads wrong block?
  - “write holes” on traditional RAIDs
    - RAID needs to write a stripe plus parity block, but doesn’t perform those updates atomically...

ZFS Software Structure

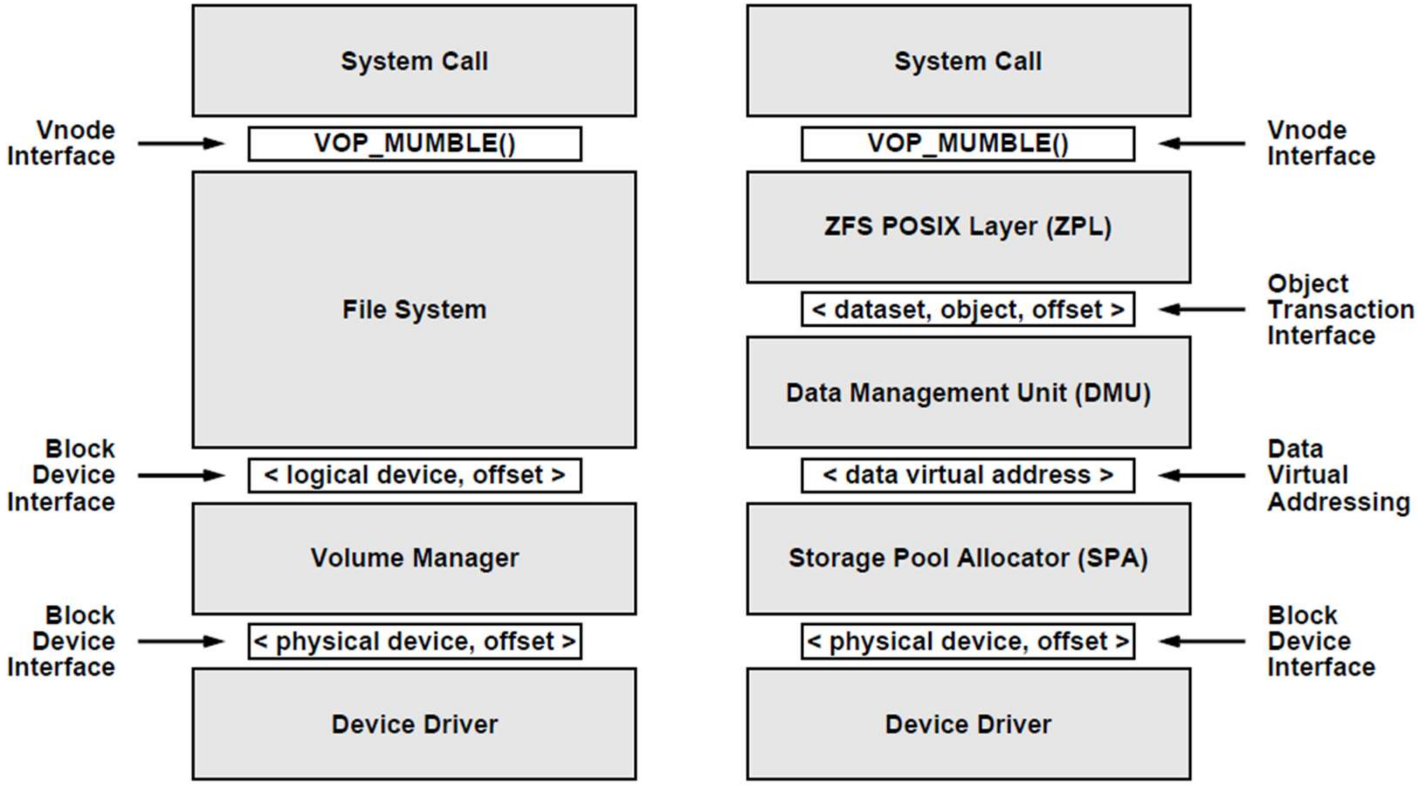


Figure 3: Traditional file system block diagram (left), vs. the ZFS block diagram (right).

ZFS Disk Management

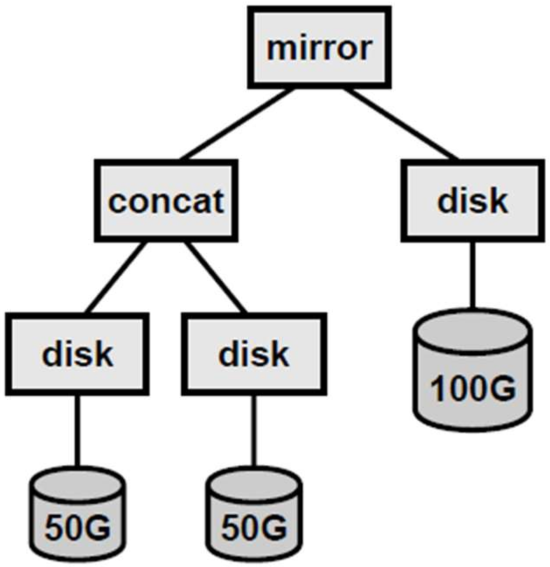


Figure 5: Example vdev with the description `mirror(concat(/dev/dsk/a,/dev/dsk/b),/dev/dsk/c)` where disks `a` and `b` are the 50 GB disks and disk `c` is the 100 GB disk.

These operations are supported in the SPA.
ZFS also implements “RAID-Z,” which is RAID-5-like but designed to be resilient to failures during write of a stripe.

ZFS error handling

- A huge file system is likely to experience errors
- "Errors" aren't just crashes
- Errors can be related to the disk (and be smaller than full device failure):
  - disk failures
  - disk read/write bit errors
  - larger disk errors (e.g., read/write wrong block)
- Errors can be file system bugs
  - You read/write the wrong thing…
- You can't fsck a huge file system
- ZFS amortizes the overhead of dealing with errors over all operation
  - extra effort is taken to detect errors "immediately" so that they're small-grained and can be fixed
- Among other things, ZFS supports a kind of mirroring at the object level (rather than the disk level -- it does disk level as well...)

- *Note: there is current interest in protecting against errors that occur in the CPU – both hardware errors (e.g., memory bit errors) and software errors (plain old bugs).*

ZFS Checksums

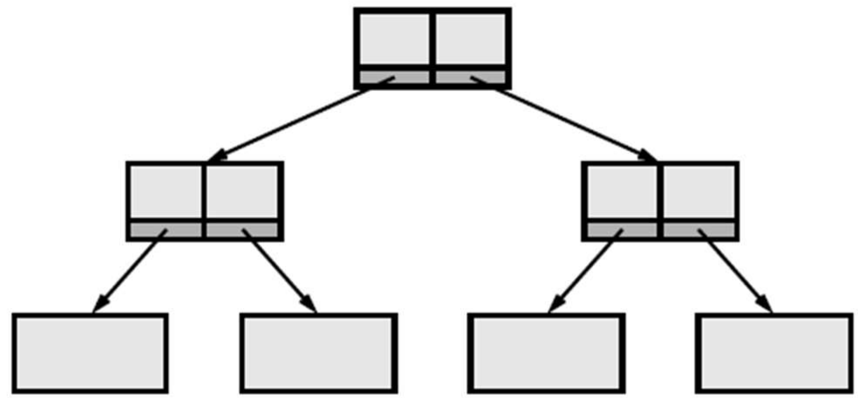


Figure 4: ZFS stores checksums in parent indirect blocks; the root of the tree stores its checksum in itself.

- Every block is checksummed
- The checksum is kept in the parent block, the one holding a pointer to the block
  - all blocks have a parent block except the “uberblock”(s)
    - the uberblock stores its own checksum
- checksums are verified whenever the block is read and recalculated whenever they’re written
- Note: disk devices do their own (sector-level) checksumming
  - this is on top of that
- Note: despite disk devices doing their own checksumming, undetected errors are observed in the field
- When a checksum error is detected, ZFS can automatically repair using one of the copies

ZFS Block Pointer

- Pointer can refer to up to 3 copies of the block
- Block size isn't fixed
- Blocks can be stored compressed
- PSIZE is physical size, LSIZE is logical size (ASIZE includes indexing overhead
- checkum[0-3] are copies of the block's checksum value
- Blocks have a type (e.g., to indicate whether it's a data block or an indirect block)

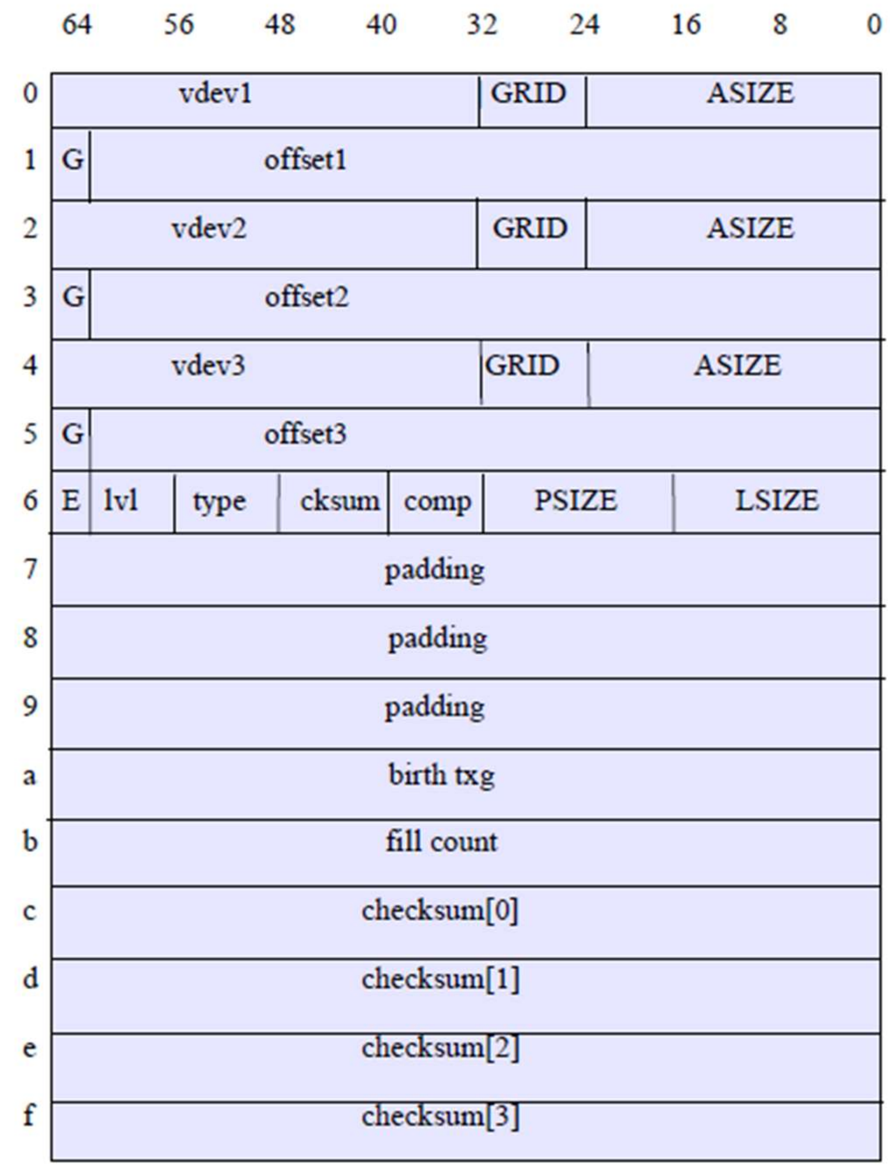


Illustration 8 Block pointer structure showing byte by byte usage.

ZFS Crash Resilience

- ZFS guarantees that the disk always contains a coherent version of the file system
- All disk writes are transactional
  - Each write is associated with a transaction group
  - A transaction group either makes it to disk in its entirety or it's as if it never existed
- However, it doesn't normally do journaling
  - So no need to process a log on reboot
- Instead, it periodically does write-back of transactions
  - Mostly they succeed, but we still need a mechanism for if they fail

ZFS Journaling

- ZFS journals in two cases

- If an app wants to synch right now, its update transaction is written to a log on stable storage
  - But its transaction is also maintained in the write-back cache
  - Usually the transaction goes to disk when periodic update occurs and then the log entry is unlinked
  - (So, mostly the log is written by never read)

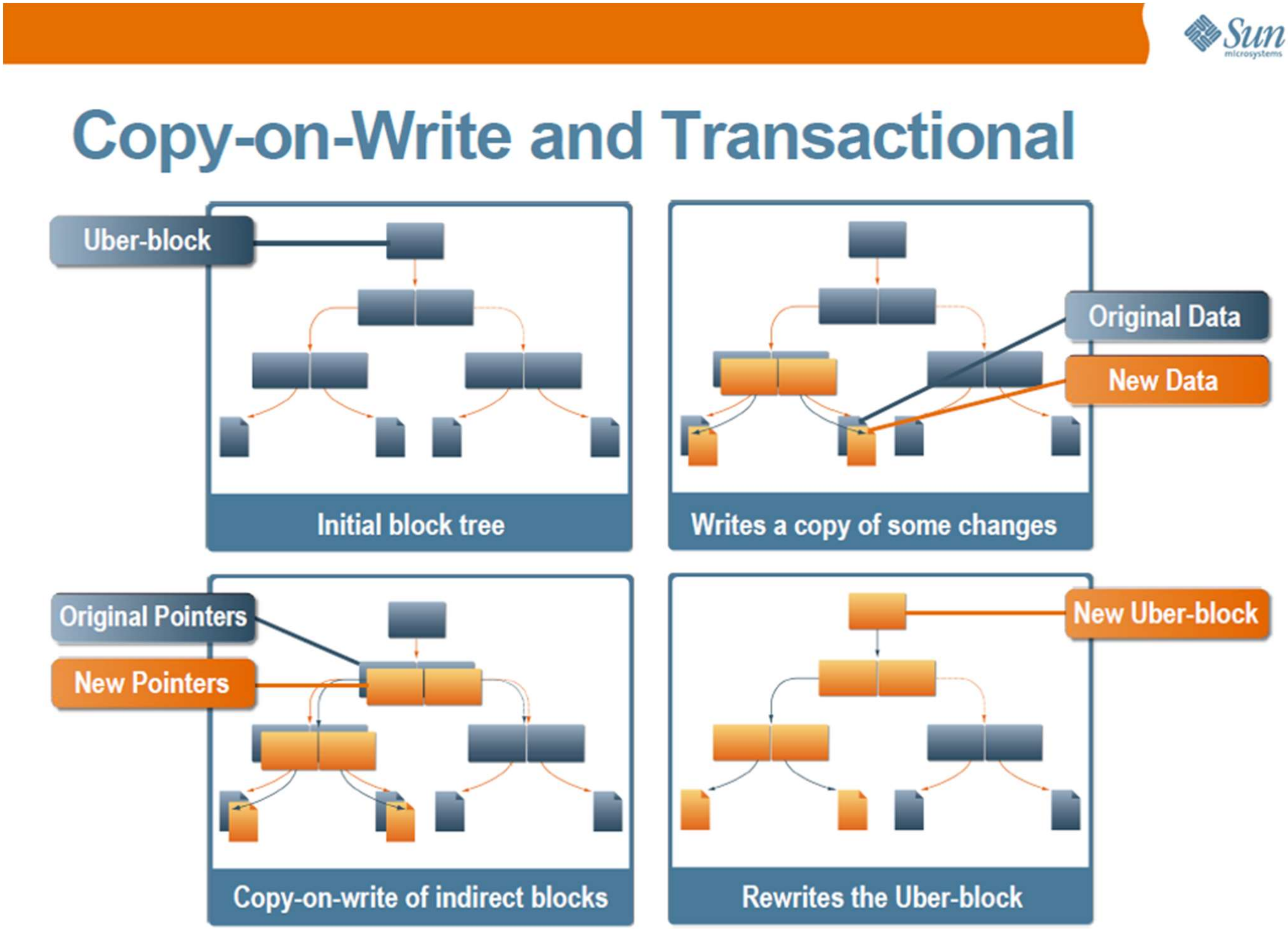
- A “Delete queue”
  - Written at the ZPL level
  - Records the intention to delete file/directories

ZFS Crash Resilience

- If ZFS isn't doing logging, how does it get transactional updates?

- What it does feels similar to the RCU (read-copy-update) lock we saw earlier
  - Copy-on-write updates of logical blocks
  - A single (hopefully) atomic operation installs a new version of the file system
    - The old version can be garbage collected, if you want
    - The old version can be maintained, as a "snapshot"

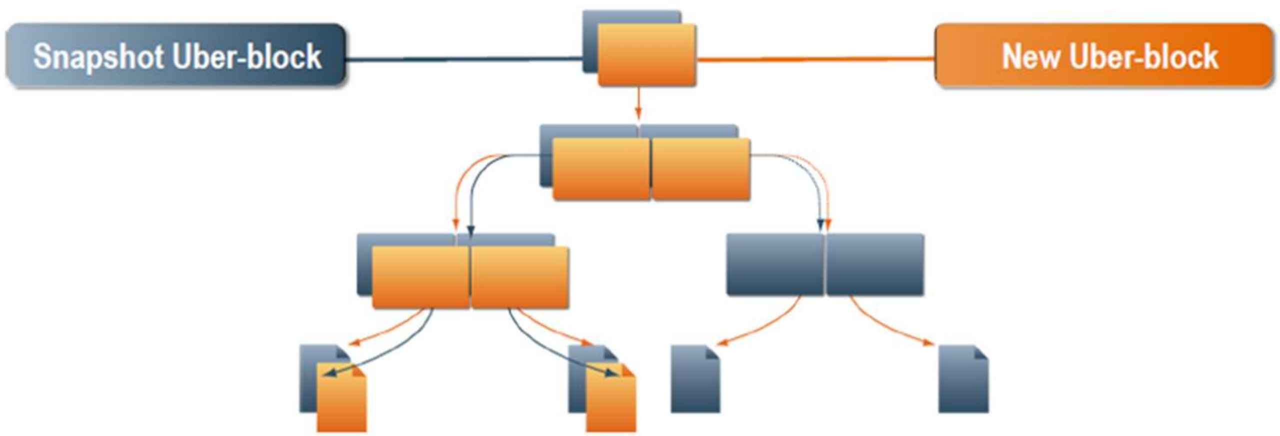
ZFS Crash Resilience



ZFS Snapshots

ZFS Snapshots

- View of a file system as it was at a particular point in time.
- A snapshot initially consumes no disk space, but it starts to consume disk space as the files it references get modified or deleted.
- Constant time operation.



The snapshot is basically a diff, so its size is related to the number of bytes changed, not the size of the entire file system.

vdev Label and Uber-blocks

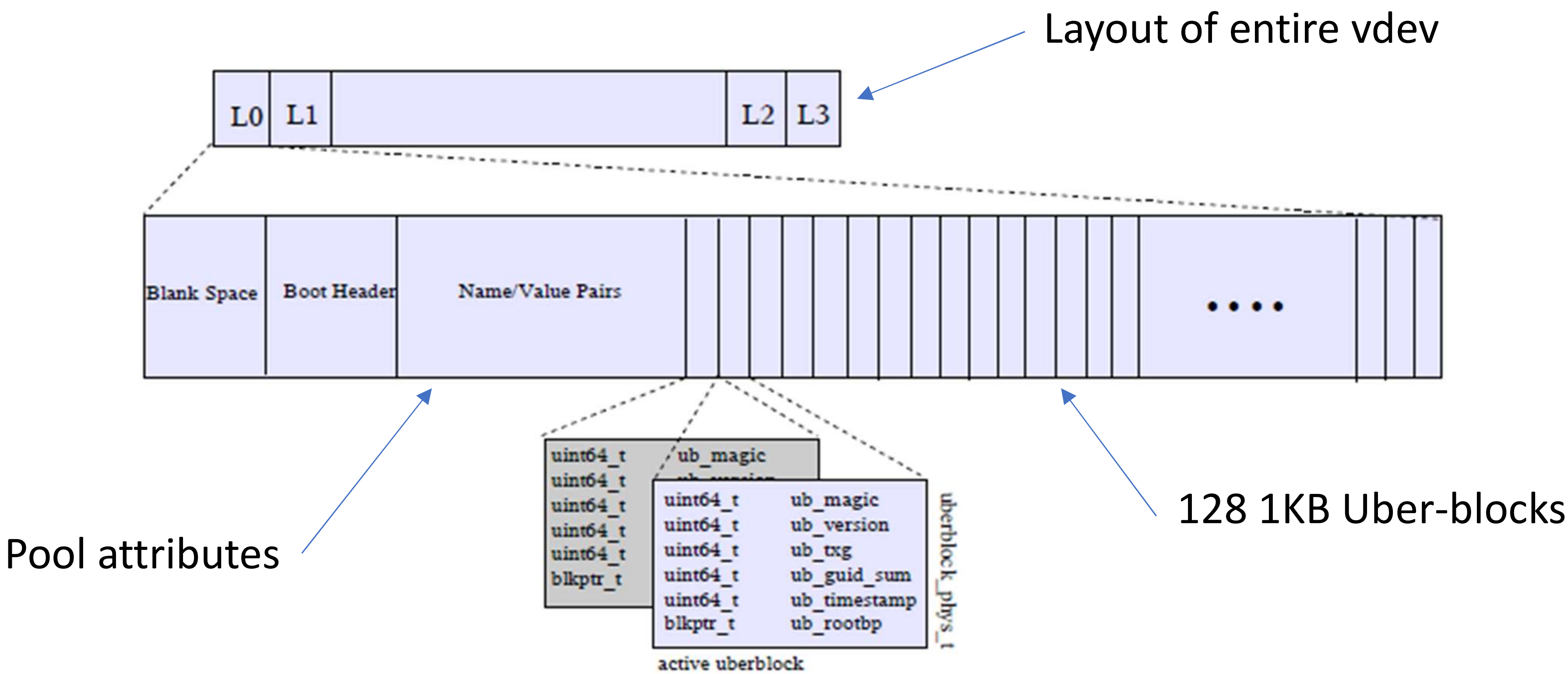


Illustration 6 Uberblock array showing uberblock contents

- Label updates first write L0 and L2 and then write L1 and L3
- Uber-block updates are written round-robin across disks
- On (re)boot, the most recently written Uber-block is made current

ZFS: File System Imposed Size Limitations

ZFS implementors wanted to accommodate exponential growth in storage capacity...

File System	Max File Size	Max Volume Size	Max # Files
FAT32	4GB	16TB	-
NTFS	16EB	16EB	2^{32}
ext4	16TB	1EB	2^{32}
ZFS	16EB	2^{78}B	2^{128}

1EB = 1,000,000 TB

ZFS Summary

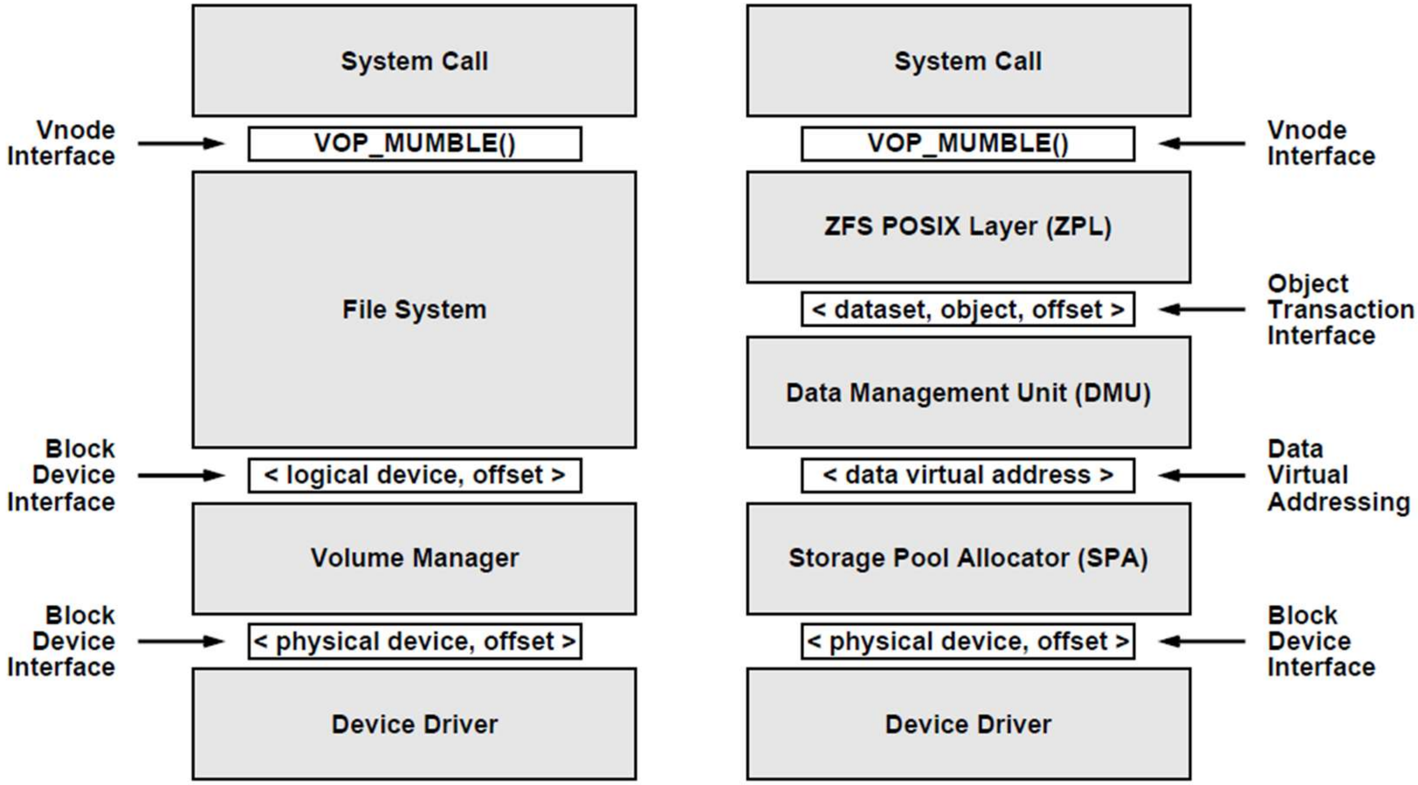


Figure 3: Traditional file system block diagram (left), vs. the ZFS block diagram (right).

Traditional Disk Storage Administration

But with ZFS....

More Information

- The paper linked from the course calendar
- The slide deck linked from the course calendar
- The Internet