ZFS UTH
Under The Hood

London Open Solaris User Group
16th May 2007

Jason Banham & Jarod Nash

Systems TSC

Sun Microsystems

ZFS I/O Stack

Object-Based Transactions

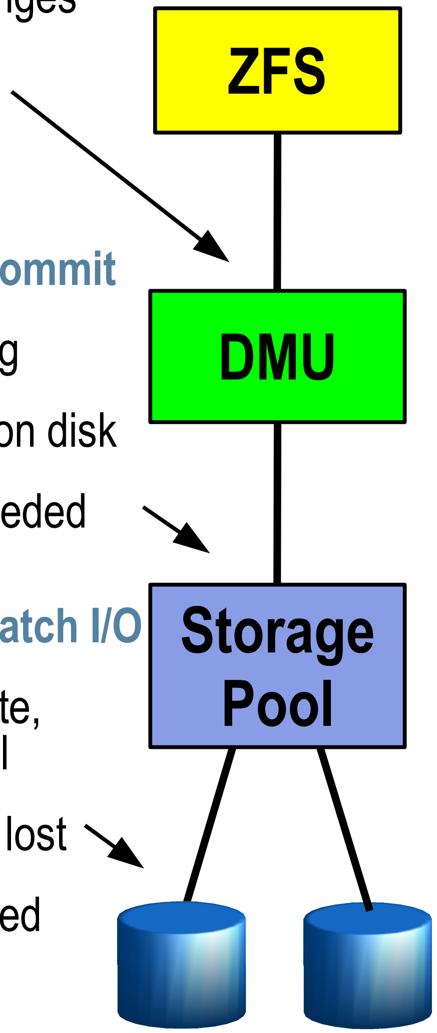
- "Make these 7 changes to these 3 objects"
- All-or-nothing

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 Under The Hood

A Few Words on the Implementation

"ZFS is about 25,000 lines of kernel code and 2,000 lines of user code, while Solaris's UFS and SVM (Solaris Volume Manager) together are about 90,000 lines of kernel code and 105,000 lines of user code. ZFS provides more functionality than UFS and SVM with about 1/7th of the total lines of code."

The Zettabyte File System
Bonwick, Ahrens, Henson, Maybee, Shellenbaum
ACM SOSP 2003

ZFS
The Layered Overview

- See laminated handout...

ZVOL, ZFS and ZPL

- ZVOL - ZFS Volume Emulator
  - Character/block device (rdsk/dsk) access to pooled storage
- ZPL (ZFS Posix Layer)
  - ZPL uses DMU objects to construct files/directories of filesystem
  - ZPL provides filesystem semantics such as ownership, permissions, and times ([acm]time + creation time)
  - ZPL implements VFS/vnode (Solaris Virtual FS layer):
    - zfs_mount(), zfs_umount(), zfs_statvfs(), zfs_sync(),...
  - Maps system calls into object based transactions
    - “Make 7 changes to these 3 objects”
  - ZFS filesystem creation consists of creating a few DMU objects
    - No newfs...more like mkdir

DMU

Data Management Unit

- DMU is the foundation of ZFS
- Provides transaction based *object* model
  - Object: blocks of storage allocated from the SPA
- Responsible for on-disk data consistency
- Consumers/ZPL use *transactions* to interact with DMU
  - Transaction (TX): Operation on an object or objset, committed as part of a transction group (TXG)
- Key DMU components:

ZAP	- ZFS Attribute Processor
DSL	- Dataset and Snapshot Layer
Transaction Engine	- the *heart* of ZFS
ZIL	- ZFS Intent Log

DMU

blkptr_t – Block Pointer

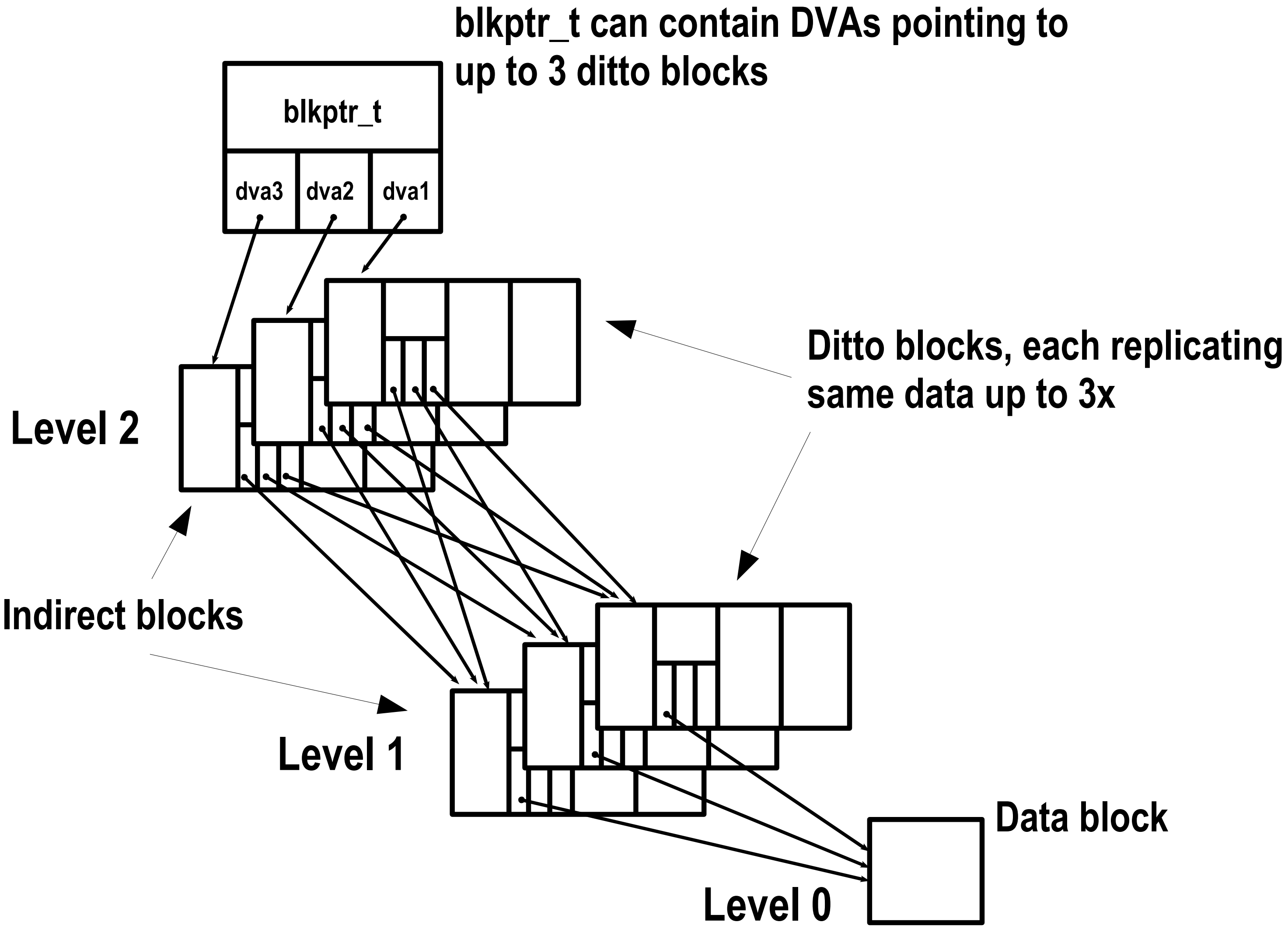
- Data is transferred between disk and main memory in units called *blocks*
- A block pointer (blkptr_t) is a ZFS structure used to locate, verify, and describe blocks of data
- blkptr_t identifies checksum, compress and endianess
  - blkptr_t also contains checksum
  - Bytes swapped on read, then written native if the pool is moved
- Size of block
  - Logical Size of data without compression, RAIDZ or Gangblocks
  - Physical Physical size of the block on disk after compression
  - Allocated Total size of all blocks allocated to hold this data including any gang headers or RAIDZ parity information

DMU
DVA – Data Virtual Address

- DVA is the combination of the VDEV and offset portions of the blkptr_t
- blkptr_t's can store up to three copies of the data pointed to by the blkptr_t, each pointed to by a unique DVA
  - The data stored in each of these copies is identical
  - The number of DVAs used per blkptr_t is a policy decision and is called the “wideness” of the blkptr_t
  - These copies are known as *ditto blocks,* by default:
    - 1 DVA for user data
    - 2 DVAs for filesystem metadata
    - 3 DVAs for metadata that's global across all filesystems in the pool
  - Approx 2% hit in terms of space/IO for this added redundancy

DMU

Ditto Blocks Diagram



blkptr_t can contain DVAs pointing to up to 3 ditto blocks

Ditto blocks, each replicating same data up to 3x

ZAP
ZFS Attribute Processor

- Routines to handle arbitrary (name, object) associations within an object set (objset)
  - Most commonly used to implement directories
  - Also used extensively throughout the DSL
  - As a method of storing pool-wide properties
    - mountpoint, compression (enabled/disabled/algorithm)
- Two types of ZAP object:
  - MicroZAP
  - FatZAP
- MicroZAPs implement a simple mechanism for storing a small number of restricted attributes, FatZAPs handle everything else

ZAP

MicroZAP vs FatZAP

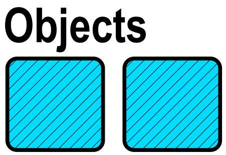
- A MicroZAP is used if three conditions are met:
  - Name portion of each attribute is less than or equal to 50 characters in length (including NULL terminating character)
  - Value portion of all attributes are of type uint64_t
  - All name-value pair entries fit into one block – max data block size in ZFS is 128Kb (allowing up to 2047 entries in a MicroZAP)
- If any of these conditions fail, a FatZAP is used
- MicroZAPs are automatically upgraded to FatZAPs

DSL

Dataset & Snapshot Layer

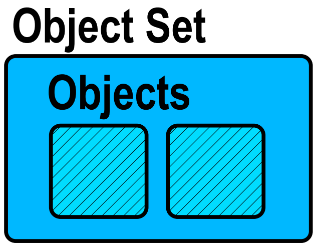
- Provides a mechanism for describing and managing relationships-between and properties-of object sets
  - > 4 types of objset: filesystem, clone, snapshot, volume
- Relationships...
  - > Relationship to parent/child filesystems
  - > Relationship with snapshots and clones
- Properties...
  - > Usage, quotas, reservations, compression in use, etc
- Relationship and Properties implemented on top of objsets as datasets and dataset directories...

DSL
Object Overview - Objects



- Objects
  - > Blocks of storage allocated from the SPA
  - > Everything in ZFS is an object
  - > A dnode describes and organizes a collection of blocks making up an object
  - > A znode is the ZPL level representation of a file/directory
  - > Notionally:
    dnode+znode ≡ UFS inode

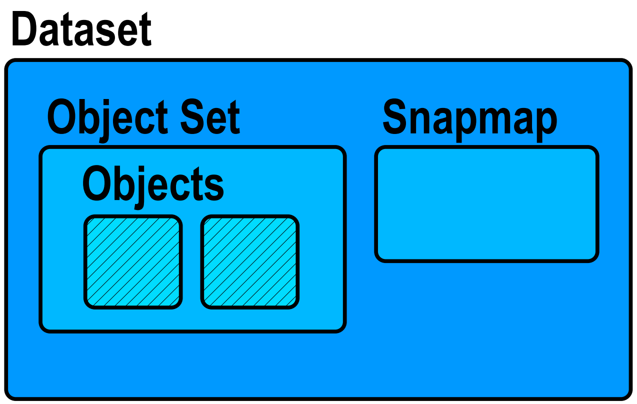
DSL
Object Overview – Object Sets (objsets)



- Objsets
  - Group related objects, such as objects in a filesystem
  - DSL provides manages relationships between objsets

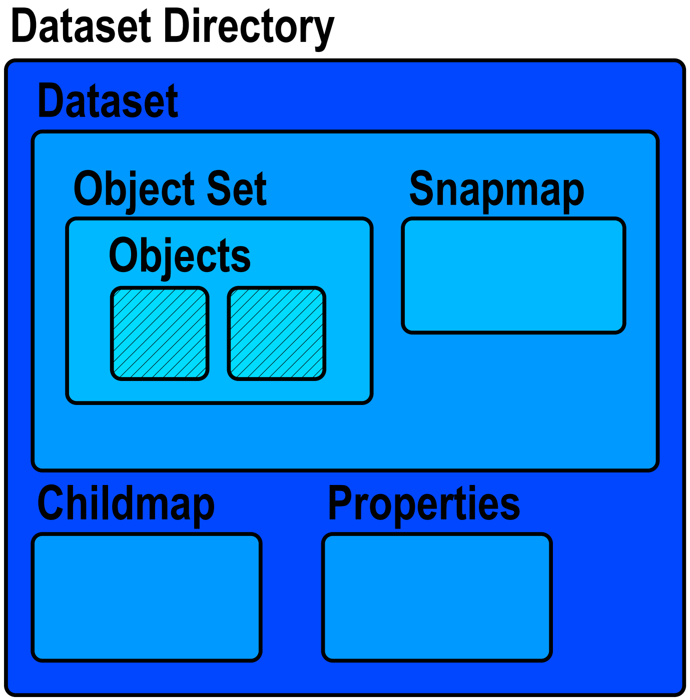
DSL

Object Overview - Datasets



- Dataset
  - > Encapsulates objset and provides
    - > Space usage
    - > Snapshot relationships

DSL
Object Overview – Dataset Directory



- Dataset Directory
  - Groups datasets
  - Properties such as quotas, reservations, compression
  - Dataset relationships
    - A child is dependent on the existence of its parent. A parent can not be destroyed without first destroying all children

Transaction Engine
Overview

- The heart of ZFS, driving I/O behaviour
- Changes to objects/objsets (transactions/TX) are grouped together as transaction groups (TXG)
- TXG updates either succeed or fail as a whole
  - Delivers on-disk data consistency
- Significant performance gains for updates
  - Removes majority of I/O ordering constraints
    - Deleting a file:
      1. Start a transaction
      2. Remove file from directory object and free the object's dnode (contains znode and file blkptr_t's), *in any order*
      3. Commit the transaction

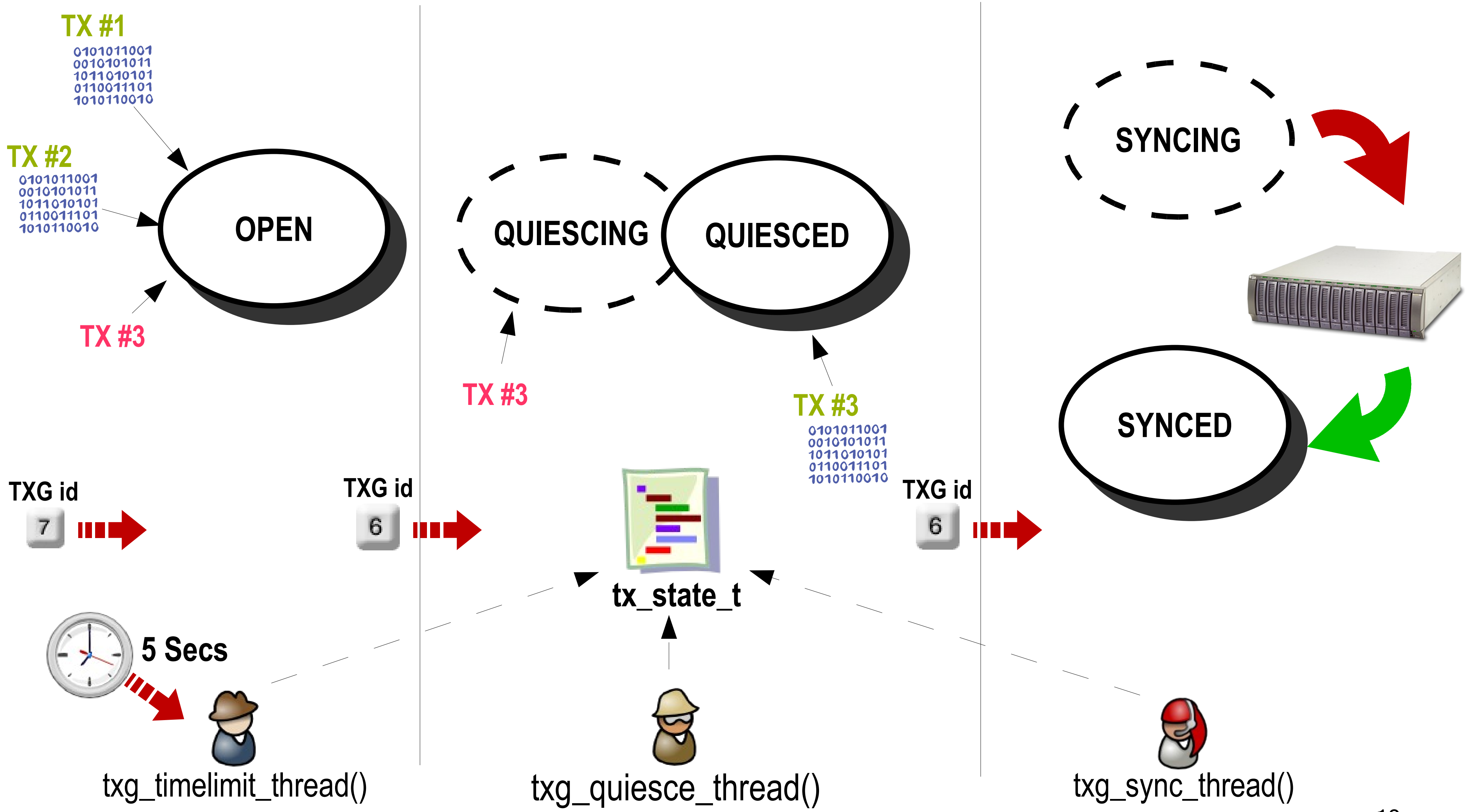
Transaction Engine
Transaction Group States

- TXGs can exist in one of three states:
  - Open - Accumulating new updates
  - *Quiescing* -> Quiesced - Waiting for updates to commit
  - *Syncing* -> Synced - Flushing updates to storage
- Consumer/ZPL use of TXs and TXGs
  - 1. Create TX
  - 2. Assign TX to a TXG
  - 3. Modify the objects as part of the TX
  - 4. Commit TX

Transaction Engine
Driver Threads

- txg_timelimit_thread
  - Fires every five seconds
  - Attempts to move the transaction engine to the next state
- txg_quiesce_thread
  - Responsible for ensuring any pending modifications to objects have completed (quiesced)
  - Wakes up the sync thread when all updates have been committed
- txg_sync_thread
  - Responsible for syncing data out to stable storage
- Single state structure is used by all three threads
  - tx_state_t: records TXG number for each state

Transaction Engine
TXG State Transistion Diagram



ZIL
ZFS Intent Log

- Filesystems buffer write requests and sync these to storage periodically to improve performance
- Power loss can corrupt filesystems and/or suffer data loss
  - Corruption solved with TXG commits
    - Always on-disk consistency
- Use *synchronous* semantics for applications requiring data is flushed to stable storage by the time a system call returns
  - Open file with O_DSYNC
  - Flush buffered contents with fsync(3c)
- The ZIL provides synchronous semantics for ZFS

ZIL
Operational Overview

- Zilogs (ZIL Logs) used to record write/modify transactions
  - Zilog has enough data to replay the transaction
- Common case: defaulting to write/modify buffering:
  - Zilogs not written to stable storage
- When synchronous semantics required:
  - Zilogs written to disk
  - Log blocks dynamically allocated/freed from available blocks
  - Disk based log is on a per dataset/filesystem basis
- Zilogs freed on TXG commit
- In the event of power failure/panic the transactions are replayed from zilogs

ARC

Overview and Purpose

- ZFS does not use page cache like UFS (except: mmap(2))
- Adaptive Replacement Cache
  - Based on Megiddo & Modha (IBM) at FAST 2003
    - ARC: A Self-Tuning, Low Overhead Replacement Cache
  - ZFS ARC differs slightly in implementation
    - ZFS: Variable sized cache and contents, non-evictable contents
- DMU uses the ARC to cache DVA data objects
- 1 ARC per system
- 2 LRU (Least Recently Used) caches plus History
  - Recency (MRU) and Frequency (MFU)
    - ARC data survives large file scan
  - 1c cache and 1c history (c = cache size)

ARC

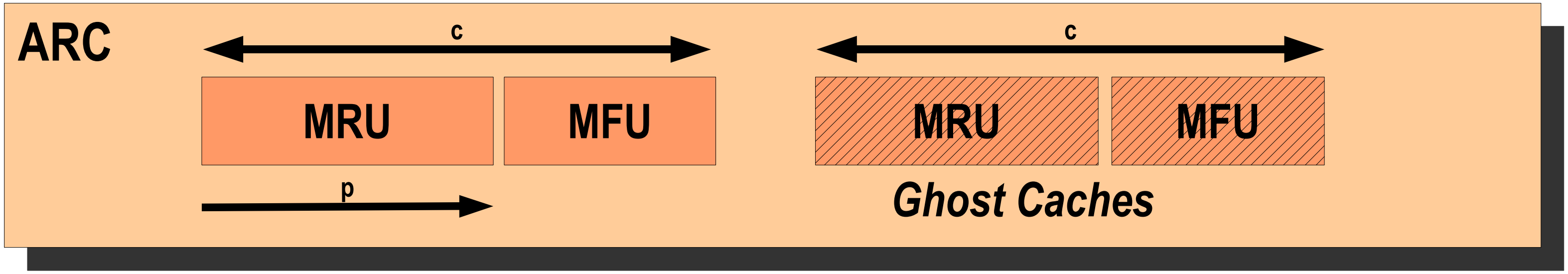
Buffer States

- ARC uses two data structures:
  - Header contains metadata (DVA, TXG birth, state, etc)
  - Buffer contains pointer to cached data
- ARC buffers are in 1 of 5 states:

anon	no DVA, dirty block copies before written out, treated as part of MRU
MRU	recently used and cached
MFU	frequently used (more than once) and cached
MRU ghost	recently used and not cached
MFU ghost	frequently used and not cached

- Ghost caches only contain ARC buffer headers

ARC
Diagram of Caches



- MRU = Most Recently Used, MFU = Most Fequently Used
- ARC adapts c and p in response to workloads
- ARC parameters initialised to:

```
arc_c_min = MAX(1/32 of all mem, 64Mb)
arc_c_max = MAX(3/4 of all mem, all but 1Gb)
arc_c = MIN(1/8 physmem, 1/8 VM size)
arc_p = arc_c / 2
```

ARC

Caches in Action

- If evicting during cache insert, then:
  - 1. Inserting in MRU & MRU < p then arc_evict(MFU)
  - 2. Inserting in MRU & MRU > p then arc_evict(MRU)
  - 3. Inserting in MFU & MFU < (c-p) then arc_evict(MRU)
  - 4. Inserting in MFU & MFU > (c-p) then arc_evict(MFU)
- Buffers change state (ie cache) in response to access
  - If current state is MRU, and at least ARC_MINTIME (62ms) since last access, then new state is MFU
  - All other repeated accesses result in state of MFU
    - Exception: Prefetching in MRU or Ghosts results in MRU

ARC
Adapting and Adjusting

- Adapting...*adapting to workload*
  - When adding new content:
    - If (hit in MRU_Ghost) then increase p
    - If (hit in MFU_Ghost) then decrease p
    - If (arc_size within (2*maxblocksize) of c) then increase c
- Adjusting...*adjusting contents to fit*
  - When shrinking or reclaiming:
    - If (MRU > p) then arc_evict(MRU)
    - If (MRU+MRU_Ghost > c) then arc_evict(MRU_Ghost)
    - If (arc_size > c) then arc_evict(MFU)
    - If (arc_size + Ghosts > 2*c) then arc_evict(MFU_Ghost)

ARC

Reclaiming

- Reclaim...*reclaiming kernel memory*
  - Every second (or sooner if adapting or kmem callback)
  - Check VM parameters: freemem, lotsfree, needfree, desfree
  - If required:
    - Set arc_no_grow – suspend ARC adaption growths
    - Set Aggressive Reclaim Policy triggers ARC shrink
      - Shrinks by MAX(1/32 of current size, VM needfree) down to arc_min
      - Calls arc_adjust() to adjust (ie evict) cache contents to new sizes
    - Call kmem_cache_reap_now() on ZIO buffers
- Megiddo/Modha said:

  "We think of ARC as dynamically, adaptively and continually balancing between recency and frequency - in an online and self-tuning fashion - in response to evolving and possibly changing access patterns"

ZIO
Overview

- All data to/from disk goes through the ZIO framework
- Responsible for translating DVAs into LBAs (Logical Block Address) on leaf VDEVs
- Multi-stage *pipeline* using a bitmask to control each stage
- Performs Checksumming and Compression as necessary
  - Encryption not implemented yet, but can accommodate
- Drives Mirroring, RAIDZ, Gangblocks and I/O retry

ZIO
Using the Pipeline

- Create a ZIO for the specific operation
  - I/Os can consist of a multiple child I/Os
    - Parent/Child dependencies are handled by the pipeline
  - ZIO creation determines the make up of the pipeline
- Start the I/O
  - Synchronous (zio_wait()) or Asynchronous (zio_nowait())
    - Use a taskq to drive asynchronous actions but may revert back to synchronous pipeline actions
- Pipeline moved along to the next stage with:
  - zio_next_stage()
  - zio_next_async_stage() - mostly asynchronous

ZIO

ZFS Read Pipeline

- Start with simple pipeline for Read:

Read Down

I/O Types	Pipeline Interlock		Checksum			Virtual Device I/O
R----	Open					
R----	Wait for children ready					
R----	Ready					
R----						I/O start
R----						I/O done
R----						I/O assess
R----	Wait for children done					
R----			Checksum verify			
R----						
R----						
R----	Done					

Read Down

- Additional optional stages based on blkptr_t bits
- Different I/O types make use of different stages

ZIO

ZFS Read Pipeline with Compression

- ZFS blkptr_t field identifies if Compression is in use
- Triggers additional Compress stage after Checksumming:

I/O Types	Pipeline Interlock	Compression	Checksum			Virtual Device I/O
R----	Open					
R----	Wait for children ready					
R----	Ready					
R----						I/O start
R----						I/O done
R----						I/O assess
R----	Wait for children done					
R----			Checksum verify			
R----						
R----		Read decompress				
R----	Done					

- Compression and Checksumming are pluggable
  > gzip compression added after ZFS initial implementation
- Checksumming can “self checksum” blocks:
  > Used for VDEV labels/Uberblock, Gangblocks and Zilogs

ZIO

ZIO Write Pipeline

- ZFS Write pipeline more complicated:

I/O Types	Pipeline Interlock	Compression	Checksum	Gang Blocks	Data Virtual Addressing	Virtual Device I/O
-W---	Open					
-W---	Wait for children ready					
-W---		Write compress				
-W---			Checksum generate			
-W---				Gang pipeline setup		
-W---				Get gang header		
-W---				Rewrite gang header		
-W---					DVA allocate	
-W---			Gang checksum generate			
-W---	Ready					
-W---						I/O start
-W---						I/O done
-W---						I/O assess
-W---	Wait for children done					
-W---	Done					

ZIO

I/O Types

- Read
- Write
- Free Free block associated with the specified blkptr_t
- Claim Used to claim blocks which the ZIL may have written as part of the intent log but which the SPA thinks are not allocated because the last TXG did not commit. Part of ZIL recovery
- Ioctl Used to issue DKIOCFLUSHWRITECACHE ioctl to flush the disk's write cache

ZIO

All ZIO Pipeline Stages (before encryption)

I/O Types	Pipeline Interlock	Compression	Checksum	Gang Blocks	Data Virtual Addressing	Virtual Device I/O
RWFCI	Open					
RWFCI	Wait for children ready					
-W---		Write compress				
-W---			Checksum generate			
-WFC-				Gang pipeline setup		
-WFC-				Get gang header		
-W---				Rewrite gang header		
--F--				Free gang members		
---C-				Claim gang members		
-W---					DVA allocate	
--F--					DVA free	
---C-					DVA claim	
-W---			Gang checksum generate			
RWFCI	Ready					
RW--I						I/O start
RW--I						I/O done
RW--I						I/O assess
RWFCI	Wait for children done					
R----			Checksum verify			
R----				Read gang members		
R----		Read decompress				
RWFCI	Done					

ZIO
Transformation Stacks

- ZIO provides a mechanism by which the data can be changed as it progresses through the I/O pipeline
- Implemented using a simple stack/linked list on the zio_t
- zio_push_transform()
  - > allocates transform buffer, new data pointers copied into ZIO and buffer, then transform buffer pushed on stack
- zio_pop_transform()
  - > pops transform buffer, frees memory, copies new top data pointers into ZIO and returns popped data
- Used for Compression, Gangblocks and Self Checksumming

VDEVs
Overview

- Implements the usual volume manager services
- 2 types:
  - > Physical or leaf (maybe whole disk, disk slice or even UFS file)
  - > Logical or interior (eg mirror)
- Modular, simple and lightweight:
  - > Logical VDEVs for Mirrors, stripes, concats
  - > Each implements a simple set of routines
- Most VDEVs take only a few 100 lines of code
  - > On-disk consistency is maintained by the DMU, rather than at the block level

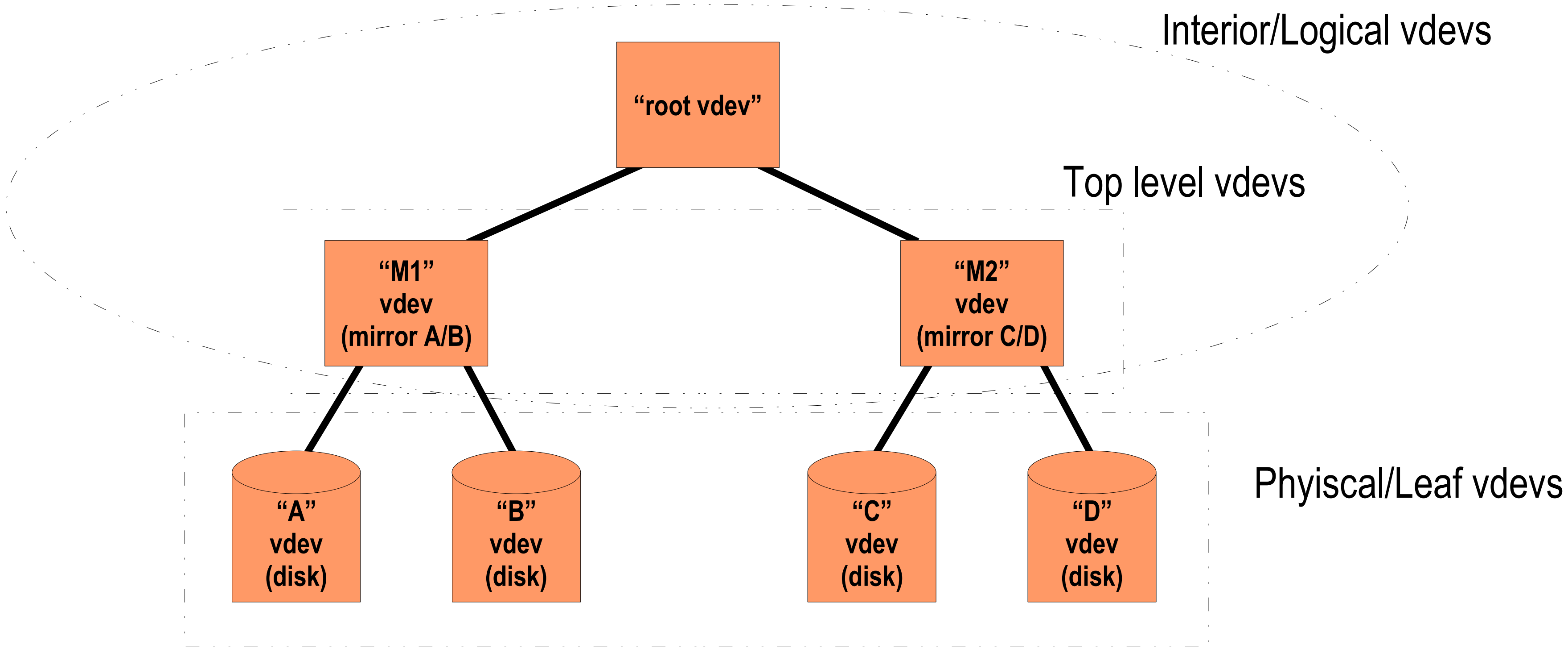
VDEVs

Modular Structure

- Each VDEV has one or more children allowing for arbitrarily complex pool configurations
- A mirror VDEV takes a write request and sends it to all children (creating new ZIOs), but it sends a read request to only one (randomly selected) child
- Similarly, a stripe VDEV takes an I/O request, figures out which of its children contains that particular block, and sends the request to that child only
- VDEVs are responsible for block allocation, tracking freespace (Spacemaps), DTL (resilvering)

VDEVs

Sample tree configuration



VDEV Labels and Uberblock

Overview and Redundancy

- Label describes physical VDEV and identifies all other VDEVs which share a common top level VDEV
- Provides access to pool, verifies integrity and availability
- 4 Identical labels placed on each physical VDEV
  - L0/L1: Start of VDEV
  - L2/L3: End of VDEV
  - Labels unique to each VDEV
  - Locations fixed
- Label accomodates VTOC/EFI disk labels
- Label contains NV pairs and array of 128 Uberblocks

VDEV Labels and Uberblock

Updating

- Only one Uberblock is active at any point in time
  - > Uberblock with the highest TXG number and valid checksum
- Active Uberblock never overwritten
  - > All updates are done by writing a modified Uberblock to another element of the Uberblock array
  - > Once updated, the TXG number and timestamps are incremented thereby making it the new active Uberblock in a single atomic action
  - > Uberblocks are written in a round robin fashion across the various VDEVs
- Two stage update:
  - > Even labels (L0/L2), then odd labels (L1/L3)
  - > Allows for filesystem recovery if system crashes during update

Metaslabs and Spacemaps
Overview

- “ZFS does for storage what VM did for memory”...Bonwick
  - > Jeff Bonwick also wrote SunOS Kernel Slab Allocator
- Leaf VDEVs are broken up into *Metaslabs*
- Freespace in a Metaslab is tracked using a *Spacemap*
- AVL trees key to Implementation
  - > Self balancing binary tree data structure
  - > Metaslabs sorted according to *weight*
  - > Spacemaps sorted according to [start, end)

Metaslabs
Policy Overview

- Tiered Allocation Policy:
  - Device Selection
    - Maximise bandwidth by spreading the load across all devices
    - More disks => more bandwidth (*Dynamic Striping*)
  - Metaslab Selection
    - Devices broken up into a number of managable *metaslabs*
  - Block Selection
    - Freespace within a metaslab is tracked using *spacemaps*
- Pluggable framework for each selection
  - TBD: New policies for differing workloads/needs

Metaslabs
Device Selection Policy

- Bias towards underutilised, ie newly added devices
- Round Robin (RR) devices every 512Kb (*empirical*)
  - > Too coarse: only get one device worth of bandwidth
  - > Too fine: no benefit of *readahead*
- RR for each ZIL (intent log) block, aka zilog
  - > We don't expect to read, but do want maximum IOPs
- TBD: Avoid slow/degraded devices
- TBD: Different Policies for:
  - > Large/sequential vs Small/random
  - > Transient (ie zilogs)
  - > dnodes (clumping for better spatial density)

Metaslabs

Metaslab/Freeblock Selection Policy

- Each leaf VDEV is divided into roughly 200 metaslabs
- Choose metaslab with highest *weight*
  - Weight metaslabs with lower LBAs (higher bandwidth)
    - Modern disks have constant bit density/angular velocity => faster
  - Weight metaslabs used before
    - Helps with keeping allocs towards outer faster regions
    - Minimise seek times
  - Weight *active* metaslabs
    - Encourages these metaslabs to be finished off
- Within metaslab, use first fit
  - TBD: Workload specific freespace selection policies