# Vlserver Memory Cache


Mark Vitale


OpenAFS Workshop 2019

19 Jun 2019

# *vlserver* performance problems?

For typical OpenAFS sites, fileservers and cache managers have the highest impact on overall cell performance; vlserver performance is close to the bottom of the list of bottlenecks.

This is not a typical site...

# Site overview

- One of the world's largest OpenAFS sites
  - ~120 cells
  - a number of RW cells
    - many regional RO cells
  - ~1300 servers
  - 140,000+ clients
  - ~40,000 containers
  - Millions of volumes
- Primary use: software distribution

# High vlserver RPC rate

- VLDB: several million volume entries
- constant VLDB updates
  - cross-cell volume replication (in-house tooling)
  - intra-cell volume replication (vos release)
  - volume housekeeping (vos move, delete, etc.)
- constant VLDB lookups
  - normal lookups
  - normal negative lookups
  - abnormal negative lookups

# The problem

- vlserver throughput bottleneck
  - Most common RPC: VL_GetEntryByNameU from cache manager
    - Average execution time 3.1 ms ~= 320 calls per second max
    - How do we know this?
      - vlserver option: -enable_process_stats
      - RPC: RXSTATS_GetProcessRPCStats
      - utility: rxstat_get_process (src/libadmin/samples)
    - At peak times, this limits performance of

# Root cause

- Lookups take too long because of excessive VLDB IO
  - average over 100 read syscalls for a normal lookup
  - even higher for negative lookups
  - discovered via additional tracing (truss/DTrace)
- Excessive IO because of scalability issues in VLDB format

# VLDB: Volume Location DataBase

- “Database” is a gross misnomer

  It’s not a true database, but a structured blob of bytes; contents are addressed by physical offset ("blockindex").

- VLDB format (version 4):
  - ubik header
  - vl header
    - version, EOF pointer, free pointer, max volid, stats, etc.
    - fileserver table
    - embedded hash tables
    - pointer to first extension block
  - extension block(s)
  - volume entries

# VLDB embedded hash tables

- Allow vlserver to find a requested volume entry without sequentially scanning entire VLDB
- Four tables in all:
  - one for volume names
  - one each for RW, RO, and BK volume ids
- Small fixed hash size – 8191 “buckets”
- Hash chains are linked via “next” blockindex pointers in each entry
- Maintained automatically as volumes are added or removed
  - New entries are inserted at the head of the chain, in the vl header

# Exacerbating circumstances

- We can’t increase the number of buckets (shorten the hash chains) without changing the VLDB format.

  1.7 million volume entries / 8191 hash buckets = 213 entries average hash chain length
- An ubik read is required to follow each entry on a given hash table chain.
- The vlserver ubik buffer pool is fixed at 150 1k ubik_pages (up to 6 entries/page)
  - optimal for sequential VLDB lookups (‘vos listvldb’)
  - easily overwhelmed by multiple parallel random lookups

# More exacerbations

- Physical VLDB IO is done via syscalls, which are thread-synchronous.
  - vlservers (1.6.x) run under OpenAFS lightweight processes (LWP), which simulate multi-threading via cooperative scheduling on a single operating system process.
  - the entire vlserver blocks all threads when any thread (LWP) must perform a physical disk read.

# “It’s worse than that, Jim”

- New volumes are inserted at the head of its hash chain.
  - Therefore, old volumes (e.g. root.afs, root.cell) tend to be near the end of each hash chain.
  - Thus, the volumes most likely to require frequent lookups are also the most expensive to lookup.
- Conclusion: vlserver lookup performance degrades significantly with VLDB size for large (>50,000 volumes) VLDBs.

# Early ideas

- Tune volume lookup cache in cache managers (`afsd –volume <nnn>`)
  - too many clients; does not address root cause
- Pthreaded ubik
  - early versions had many severe problems; now stable in 1.8.x series
- mmap the VLDB
  - judged unlikely to be accepted upstream
  - reduces but does not eliminate high syscall overhead and single-threading
- Load entire VLDB into existing ubik buffers
  - lots of unknowns; never prototyped or researched further
- Optimize hash chain contents by moving frequently requested volumes volumes toward the head of the hash chain
  - some limited improvement possible; does not address root

# Proposed solution

- Use in-memory hash tables to cache information from the on-disk hash tables
  - Only chase the on-disk hash chains once
  - cache the blockindex for each volume
- don’t prescan VLDB to preload cache at restart
  - too slow – need fast turnaround on restarts

# Hash algorithm requirements

- high load factor
- hash chains as short as possible
- Reasonable performance and scalability for common operations: insertion, deletion, lookup
- avoid runtime rehash/resize

# Cuckoo hashing

- Distinctives
  - Hash table split into two (or more) partitions, each with its own independent hash function
  - fixed size and slots - no hash chains
  - "cuckoo" eviction
    - *The cuckoo does not build its own nest, but instead evicts the eggs from the nests of other birds and substitutes its own.*
- Insertion algorithm:
    - Hash and insert into any empty slot in the appropriate bucket in first partition.
    - If no empty slots, try again for second partition.
    - If still no empty slots, choose an evictee slot (LRU) and insert new entry there.
    - Repeat insertion with the former contents of the evictee slot.
    - A loop limit prevents endless insertion; when the limit is hit, the last “egg” is effectively evicted from the cache.

# Cuckoo hashing pros and cons

- Advantages
  - Good performance
    - Space (memory)   very high load factor before resize needed
    - Time (cpu)    predictable, well-behaved insertion & lookup order (big-O)
  - Runtime rehash/resize is optional
- Disadvantages
  - not well known
  - not already in OpenAFS tree

# Cuckoo hashing papers

- Rasmus Pagh and F. Rodler. Cuckoo Hashing. Journal of Algorithms 51 (2004), p 122-144.
- Rasmus Pagh. Cuckoo Hashing for Undergraduates.  Lecture at IT University of Copenhagen, 2006.
- Eric Lehman and Rina Panigrahy. 3.5-Way Cuckoo Hashing for the Price of 2-and-a-Bit. Conference: Algorithms - ESA 2009, 17th Annual European Symposium, Copenhagen, Denmark Proceedings.  DOI: 10.1007/978-3-642-04128-0_60 · Source: DBLP

# vlserver implementation

- two cuckoo hash tables
  - one table for volume names
  - one unified table for RW/RO/BK volume ids
- each table has 2 partitions
- each partition has configurable number of buckets
  - `vlserver -memhash-bits <log2(entries)>`
- each bucket has configurable number of ‘slots’
  - `vlserver -memhash-slots <slots>`
- instrumentation & debugging
  - `vos vlmh-stats [options]`
  - `vos vlmh-dump [options]`

# vlserver negative cache

- Optional set of cuckoo hash tables for negative lookups, i.e. VL_NOENT "volume not in VLDB"
  - one table for volume names
  - one unified table for volume ids (RW, RO, BK)
- Requires positive cache
- Size computed from specified # of entries:
  - `vlserver –negcache <#entries>`

# Operation

- Reads
  - Each positive or negative lookup is automatically cached in the appropriate table.
- Writes (vos volume operations)
  - New, changed, or deleted entries never modify the positive cache because the commit may fail; instead, entries are deleted when detected invalid on the first subsequent read (“lazy” invalidation).
  - However, writes MUST immediately invalidate any affected negative cache entry on the syncsite and all non-sync sites.
- Synchronization events
  - All caches are invalidated when the database is replaced on a given server.

# Results

- At least 40x real-world improvement in vlserver read (lookup) throughput
- Vlserver throughput is no longer the limiting bottleneck during peak cell loads

# Futures

- upstreaming

This slide intentionally left blank