Breaking the Metadata Bottleneck: the Exascale Filesystem DeltaFS as a LANL and Carnegie Mellon Collaboration

Qing Zheng
Carnegie Mellon University
Breaking the Metadata Bottleneck:
The Exascale Filesystem DeltaFS as a LANL and CMU Collaboration

Qing Zheng
Chuck Cranor, Greg Ganger, Garth Gibson, George Amvrosiadis
Bradley Settlemyer†, Gary Grider†
Carnegie Mellon University
†Los Alamos National Laboratory
Everyone Loves Fast Storage

DeltaFS: 20,000x faster than FS today

Image from http://esp.igpp.ucla.edu illustrating earth’s magnetic field under the influence of the solar wind.
Everyone Loves Fast Storage

DeltaFS: 20,000x faster than FS today

How long does it take to insert 2 trillion particle files into a fs directory?

57 days

OR

2 mins
Existing FS uses Dedicated Resources

Figure shows CMU’s NASD (OSD) design (now ANSI T10), root of many today’s distributed filesystem designs.

2019 Storage Developer Conference © Carnegie Mellon University. All Rights Reserved.
MDS often a Bottleneck

It could take FOREVER to finish all metadata ops
MDS often a Bottleneck

It could take FOREVER to finish all metadata ops

57 days
Common Ways for Stronger MDS

A) Better Representation

B) Better Namespace Partitioning

C) Deeper Layering
We Could Build Something Like This

Namespace spread across 2 servers

MDS

LSM

MDS

LSM

LSM-Trees for high write throughput

A caching tier for fast reads

“MDS 2.0”
Might work but would be **EXTREMELY INEFFICIENT** in delivering 1 trillion file creates in 2 mins.

Need 800 servers if each can do 10 million file creates/s.
Budget is Fixed for Each Machine

More MDS nodes means less compute nodes
MDS not busy all the time
Budget is Fixed for Each Machine

We blame the bar that separates the nodes

A waste: unable to use MDS nodes to run jobs
A much bigger waste: unable to utilize compute nodes to process metadata
A BOLD idea: having filesystems run directly on job nodes (DeltaFS)
Today: A Dedicated MDS Per Machine

A shared namespace

Job1

Job2

MDS

Persistent state

Img

2019 Storage Developer Conference © Carnegie Mellon University. All Rights Reserved.
Better: Dynamically Instantiating MDS for Jobs
Immediate Benefits from No Dedicated MDS

Simplified cluster design
No need to pool resources for MDS during cluster planning

No false sharing
My cache entries do not get invalidated by someone else’s activities

Highly agile scalability
Larger jobs can devote more resources to MDS

Better resource utilization
Would-be idle CPU cycles can be utilized to process metadata
Does this really work for my applications?
Three Types of Interaction

**No sharing**
Different jobs access different sets of files

**Sequential sharing**
One job’s output is another job’s input

**Concurrent sharing**
Multiple jobs read & write a same set of files

Works trivially today: 1 dedicated MDS, 1 global namespace

*But a global namespace is not always required for existing jobs to work*
Unrelated Jobs Do not Have to See Each Other’s Data
Concurrent Sharing? Connect to the Leader

One use case: user monitoring such as “ls -l” & “tail -F”
Need Another Job’s Data? Just Mount it & Carry on

Job1

Underlying Storage

Job2

UNMOUNT

MOUNT

Img
Mount Many If Necessary

Underlying Storage

Job1

Job2

Job3

MOUNT 1

MOUNT 2

Img1

Img2
A namespace is as good as a global namespace if a job sees all related data.
Re-imagining filesystems for future
# Machine-Oriented v.s. Job-Oriented

## A component of a machine
- Always ON, centralized
- Uses a fixed set of dedicated nodes
- Long-standing
- Accessible from every node of a machine
- A shared FS image per machine
- Runs background activities (e.g., reorganizing indexes for fast reads)
- One piece of code

## A component of a running job
- Dynamically instantiated by jobs
  - **Highly agile**: scales with job allocations
  - **Transient**: lives within a job
  - **Private**: accessed only by a job
  - **No false sharing**: one per job
- **No jitters**: all background FS work is scheduled by jobs
- **Software-defined**: code optimized for the work at hand
Machine-Oriented v.s. Job-Oriented

A component of a machine

Always ON, centralized
Uses a fixed set of dedicated nodes
Long-standing
Accessible from every node of a machine
A shared FS image per machine

Runs background activities (e.g., reorganizing indexes for fast reads)
One piece of code

A component of a running job

Dynamically instantiated by jobs
Highly agile: scales with job allocations
Transient: lives within a job
Private: accessed only by a job
No false sharing: one per job

No jitters: all background FS work is scheduled by jobs
Software-defined: code optimized for the work at hand
Decoupling MDS from the Machine

Each job can be viewed as a process group
A group of processes self-found their MDS service

One option: MDS runs as a separate job process
Decoupled from the machine

Another option: MDS runs as library within processes
Again, decoupled from the machine
Transient Service, Persistent Data

When a job ends, its FS “service” goes with it

*Data stays in the underlying storage*
Each Job Acts as a Function

1. Takes one or more FS images as **input**

2. Creates a new FS image as **output**

3. **No side effect**
Log-Structured: Each Job Appends Changes to a Log

Keeping input **immutable** so that they can be shared in a scalable way

Each FS image essentially a *pointer* to a logical log
Turtles All the Way Down

Reading from an FS image is searching through a DAG of "Δ"s

Resolve conflicts using a **job-specified ordering**

Merging & flattening for fast reads via **log compaction**
User Pays for Speed *(by Scheduling Log Compactions)*

Log compaction reduces search depth & reclaims space
Often time-consuming

Traditional: done by a dedicated MDS
*Jitters* or *wasted work*

Better: explicitly scheduled by apps
*Predictable* high performance

I’ll handle everything: “meh” speed for everyone

Optimized for bulk insertion

Fast reads

2019 Storage Developer Conference © Carnegie Mellon University. All Rights Reserved.
How does my job find its input data?
It’s All about Mapping Names to Data

User specifies names; a mechanism handles the mapping

The good old days: a job control system does the mapping

Today: a global filesystem namespace does the mapping

LANL’s Cray-1 (left) and Trinity computer (right), https://www.lanl.gov/asci/platforms/index.php
A New Kind of Mapper: Filesystem Image Registry

Works like github.com, jobs “git-clone” their input datasets

One FS image registry (can be many) → A manifest, not the image itself → Access

1. Publish

Job 1

2. Get info by name

Job 2

Publication & collection may be automated by workflow engines
Which Registry did I Use? Ask a Catalog Service

Is it github.com or bitbucket.org?

A catalog server (again, can be many)

1 Subscribe

2 Search

Job

3 Get info

Related talk: LANL’s catalog service GUFI by Dominic Manno

Session 63, 2pm Wed, Lafayette room
Sounds Good. *Remind me Why Perf. is Better*...

1. **More CPUs**
   Able to use more resources to do FS work

2. **More Efficient**
   No false sharing, less synchronization, better caching

3. **Software-Defined**
   Smart clients, simple storage
Example: Making a Needle-in-a-Haystack Hero

A job using 100K CPU cores w/ an embedded FS

12 billion file inserts/s

Up-to 5000x faster queries than bulk scans

Under the hood: a) leveraged idle CPU cycles, b) deep writeback buffering, c) optimized storage layout
Conclusion

Existing FS clients sync too often with servers
Synchronization of anything global should be avoided at extreme scales

Removing servers forces us to review what’s necessary
Enabling sequential sharing is where filesystems shine

Need radically different models for shared storage
A job-oriented filesystem scales better in many computing scenarios
Thank you.