BlobSeer: Towards efficient data storage management on large-scale, distributed systems

Bogdan Nicolae

University of Rennes 1, France
KerData Team, INRIA Rennes Bretagne-Atlantique
PhD Advisors: Gabriel Antoniu and Luc Bougé

December 1, 2010

Outline

- Context
- Related work and its limitations
- Contribution: BlobSeer
  - Principles
  - High level description
  - Zoom on metadata management
  - Synthetic benchmarks
- Applications
  - BlobSeer as a storage backend for Hadoop MapReduce
  - BlobSeer providing virtual machine image storage for clouds
  - BlobSeer as a QoS enabled storage service for applications running on the cloud

We live in exponential times...

- Every two years the amount of information doubles
- Making something useful out of it becomes incresingly difficult
- We depend more and more on large-scale computing infrastructure

Dealing with information overload: Enterprise datacenters

- Tens of thousands of machines in huge clusters
- Leveraged directly by the owner
- Commodity hardware: minimizes per unit cost
- Easy to add, upgrade and replace
- Data-intensive applications

Dealing with information overload: Clouds

- Computing as utility rather than capital investment
- Driven by pay-as-you-go model
- Several levels of abstraction: IaaS, PaaS, SaaS
- Several advantages: low entry cost, elasticity, rapid development

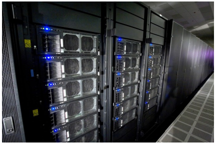
amazon web services™

NIMBUS

OpenNebula.org

Dealing with the information overload: HPC infrastructures

- Complex scientific and engineering applications
- High-end hardware
- Manipulate information at petabyte-scale and beyond



BLUE WATERS
SUSTAINED PETASCALE COMPUTING

Data storage and management is a key issue

Requirements

- Easy manipulation of data
- High access throughput
- Scalability
  - Data
  - Metadata

Current approaches: Parallel file systems

- Mostly used in HPC infrastructures
- POSIX access interface
- Data striping
- Advanced caching

lustre
Scalable Storage

ceph
PETABYTE SCALE STORAGE

Pros

- Distributed data
- MPI optimizations

Cons

- Locking-based
- Too many small files

Current approaches: Data-intensive oriented file systems

- Huge files
- Writes at random offsets are seldom
- Files grow by atomic appends
- Fine grain concurrent reads

Pros

- No locking
- Data location aware

Cons

- Centralized metadata
- Expensive updates

Current approaches: Cloud data storage services

- Virtualize storage resources
- Pay for duration, size and traffic
- Flat naming scheme
- Simple access model

amazon
webservices™
Simple Storage Service

Windows Azure

Pros

- High data availability
- Versioning

Cons

- Limited object size
- Limited concurrency control

Limitations of existing approaches

Issue	Parallel FS	Data-intensive FS	Cloud store
Too many small files	✗	**Addressed**	✗
Centralized metadata	**Addressed**	✗	**Addressed**
No versioning support	✗	✗	**Addressed**
No fine grain writes	**Addressed**	✗	✗

Limitations of existing approaches

Issue	Parallel FS	Data-intensive FS	Cloud store	???
Too many small files	✗	**Addressed**	✗	**Addressed**
Centralized metadata	**Addressed**	✗	**Addressed**	**Addressed**
No versioning support	✗	✗	**Addressed**	**Addressed**
No fine grain writes	**Addressed**	✗	✗	**Addressed**

Contribution:

BlobSeer

Data Sharing at Large Scale

Principles

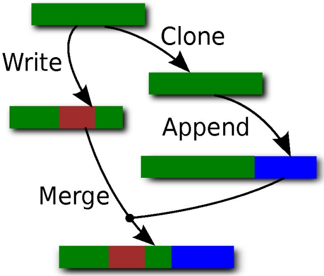
- BLOBs
  - Eliminate need to keep many small files
  - Provide fine-grain R/W access
- Data striping
  - Distributes I/O workload
  - Enables user to configure distribution strategy
- Decentralized metadata
  - Distributes metadata at fine granularity
  - Brings scalability and high avalability
- Versioning is a key design principle

Versioning as a key principle

- Clients mutate BLOBs by submitting diffs
- A BLOB is never overwritten: a new snapshot is generated
- Only diffs are stored
- Clients see whole, fully independent snapshots
- Fine-grain read access to any past snapshot is possible

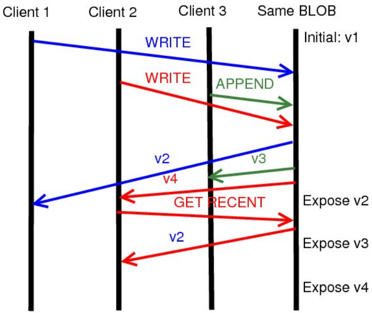
Contribution: a versioning-oriented access interface

- id = CREATE()
- v = APPEND(id, size, buffer)
- v = WRITE(id, offset, size, buffer)
- (v, size) = GET_RECENT(id)
- READ(id, v, offset, size, buffer)
- new_id = CLONE(id, v)
- dv = MERGE(sid, sv, soffset, ssize, did, doffset)



Consistency semantics

- Writes: are atomic and totally ordered
  - No guarantee when they become visible to clients
  - Finish before becoming visible to clients
- Reads: require a version explicitly
  - GET_RECENT does not guarantee latest version
  - Can read any version older than GET_RECENT



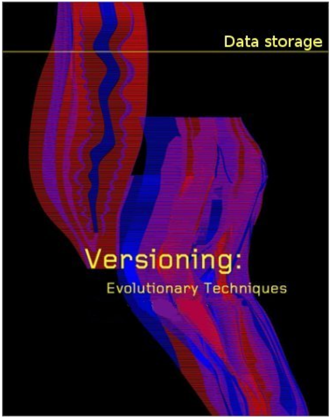
Read exposed writes only

Advantages of this proposal

- Access to historic data
- Revert to previous snapshots
- Track changes to data

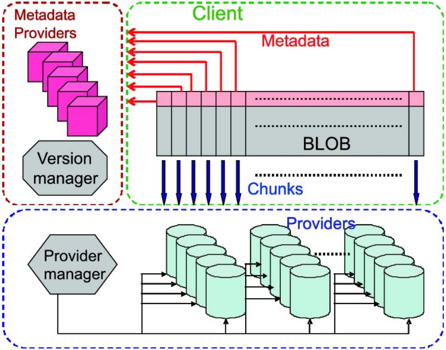
Exploit data parallelism better

- Avoid synchronization to achieve better throughput
- Build complex workflows



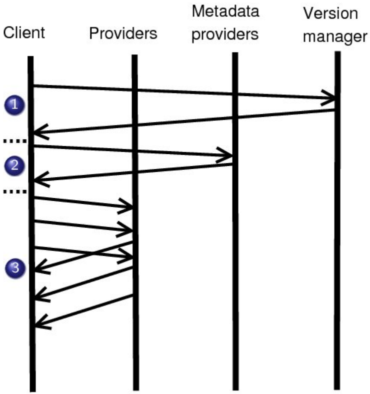
Architecture

- Data providers
- Metadata providers
- Provider manager
    - Allocation strategy
- Version manager
    - Guarantees total ordering and atomicity



How does a read work?

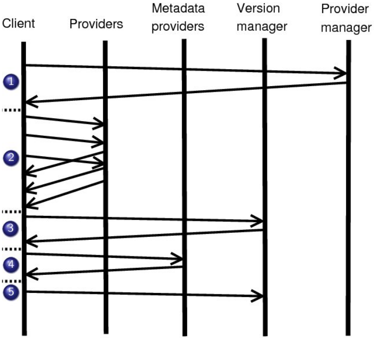
1. Select a version (optionally ask version manager for the most recently exposed version)
2. Fetch the corresponding metadata from the metadata providers
3. Contact providers in parallel and fetch the chunks into the local buffer



How does a write work?

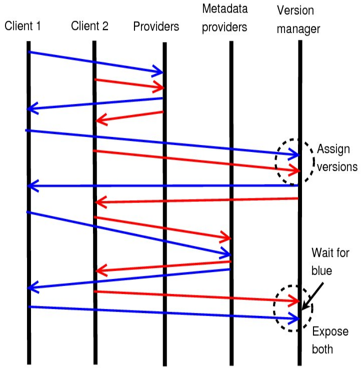
1. Get a list of providers, one for each chunk
2. Contact providers in parallel and write the chunks
3. Get a version number for the update
4. Add new metadata to consolidate the new version
5. Report the new version is ready

→ Version manager will eventually expose it



How to guarantee total ordering and atomicity?

- Clients ask for a version number
- Version manager assigns version numbers
- Clients write metadata concurrently
- Clients confirm completion
- Version manager exposes versions in order of assignment



Zoom on metadata management: Motivation

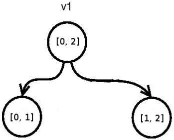
- Present fully independent snapshots in spite of writing only diffs
- Access performance should not degrade with increasing number of diffs
- Proposed so far: B-Trees, Shadowing
  - Difficult to maintain in a distributed fashion
  - Expensive synchronization for concurrent updates
- Our goals:
  - Easy to manage in a distributed fashion
  - Efficient concurrent updates

Contribution: Versioning over Distributed Segment Trees

- Deals with distributing the metadata while avoiding expensive management
- Binary tree is associated to each BLOB snapshot
- Reads descend towards leaves, writes build new trees bottom-up

Key ideas

- Keep metadata immutable
- Share whole sub-trees

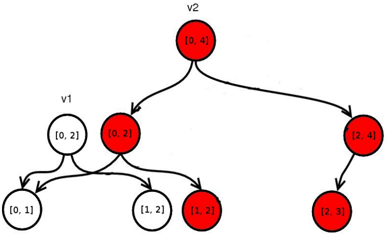


Contribution: Versioning over Distributed Segment Trees

- Deals with distributing the metadata while avoiding expensive management
- Binary tree is associated to each BLOB snapshot
- Reads descend towards leaves, writes build new trees bottom-up

Key ideas

- Keep metadata immutable
- Share whole sub-trees

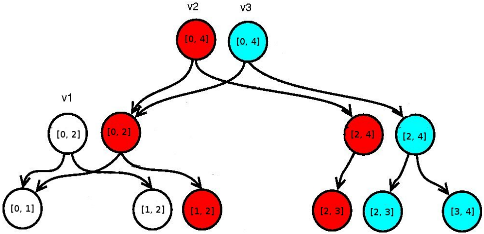


Contribution: Versioning over Distributed Segment Trees

- Deals with distributing the metadata while avoiding expensive management
- Binary tree is associated to each BLOB snapshot
- Reads descend towards leaves, writes build new trees bottom-up

Key ideas

- Keep metadata immutable
- Share whole sub-trees

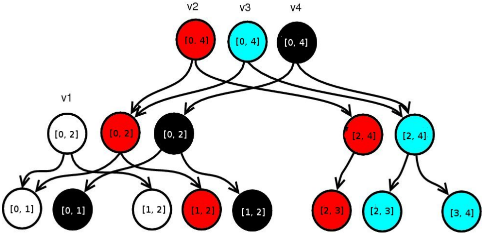


Contribution: Versioning over Distributed Segment Trees

- Deals with distributing the metadata while avoiding expensive management
- Binary tree is associated to each BLOB snapshot
- Reads descend towards leaves, writes build new trees bottom-up

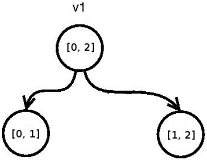
Key ideas

- Keep metadata immutable
- Share whole sub-trees



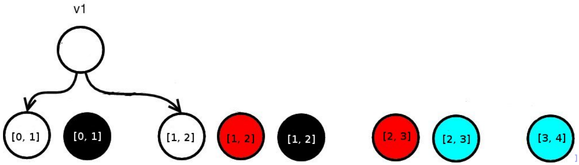
Metadata forward references

- Solve the problem of efficient concurrent updates to the metadata
- Key idea: precalculate children of lower versions instead of waiting
- Example
  - Initial BLOB
  - Three concurrent writers finished writing their chunks
  - Black is faster but knows about blue and links to the not-yet-existing node
  - After red and blue finish, metadata is consistent



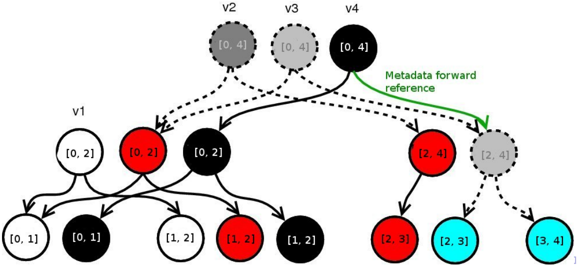
Metadata forward references

- Solve the problem of efficient concurrent updates to the metadata
- Key idea: precalculate children of lower versions instead of waiting
- Example
    - Initial BLOB
    - Three concurrent writers finished writing their chunks
    - Black is faster but knows about blue and links to the not-yet-existing node
    - After red and blue finish, metadata is consistent



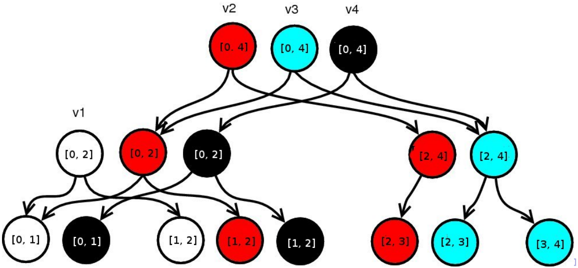
Metadata forward references

- Solve the problem of efficient concurrent updates to the metadata
- Key idea: precalculate children of lower versions instead of waiting
- Example
  - Initial BLOB
  - Three concurrent writers finished writing their chunks
  - Black is faster but knows about blue and links to the not-yet-existing node
  - After red and blue finish, metadata is consistent



Metadata forward references

- Solve the problem of efficient concurrent updates to the metadata
- Key idea: precalculate children of lower versions instead of waiting
- Example
  - Initial BLOB
  - Three concurrent writers finished writing their chunks
  - Black is faster but knows about blue and links to the not-yet-existing node
  - After red and blue finish, metadata is consistent



Design considerations

- Event-driven, layered design
  - Callbacks instead of blocking
  - Asynchronous RPC for interprocess communication
- Metadata providers form a DHT
  - Custom implementation
- Plugin-able allocation strategy on provider manager
  - Round-robin load-balancing by default
  - More adaptive solutions possible

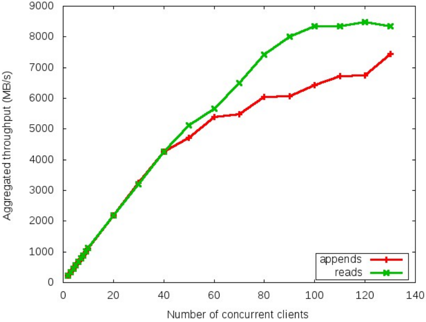
Fault tolerance

- Clients die during write
    - Before asking for a version: no problem
    - After asking for a version: delegate to metadata provider
- Data and/or metadata providers die
    - Replication of chunks and metadata pieces
    - Data and metadata immutable: no sync between replicas needed
- Version manager and/or provider manager dies
    - Distributed state machine using leader election

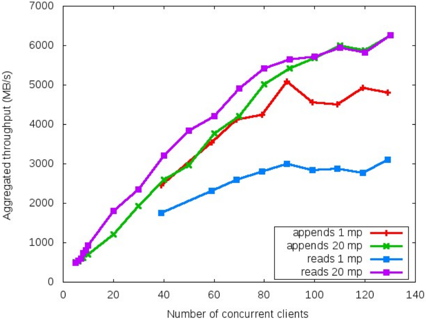
Experimental platform: Grid'5000

- Experimental testbed distributed in 9 sites around France
- A total of more than 5000 cores
- Reservation system grants exclusive access for experiments
- x86_64 CPUs, >2 GB RAM, locally attached disks
- Interconnect: Gigabit Ethernet, Myrinet, Infiniband

Results: synthetic benchmarks

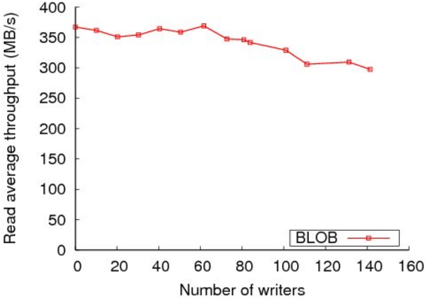


Data striping

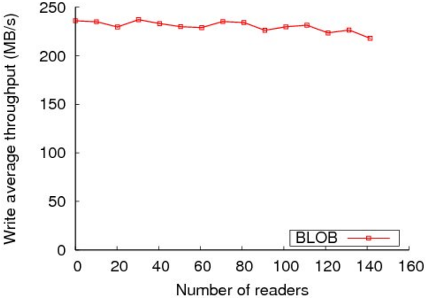


Metadata decentralization

Results: synthetic benchmarks (2)



Increase write pressure while reading



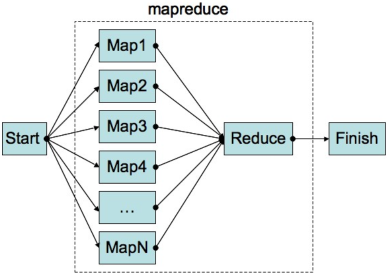
Increase read pressure while writing

Applications

- Storage backend for Hadoop MapReduce
- Efficient VM image deployment and snapshotting on clouds
- QoS enabled storage for clouds

MapReduce

- Data-intensive oriented paradigm
- Covers a wide range of data-intensive application classes
- Users stick to a well-defined model
- Widely adopted: Google, Yahoo
- Popular open-source implementation: Hadoop



BlobSeer as a storage backend for Hadoop MapReduce

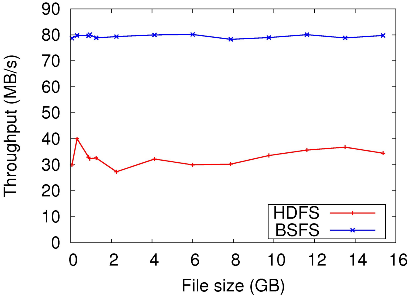
Proposal

- BlobSeer replaces HDFS (default storage backend)

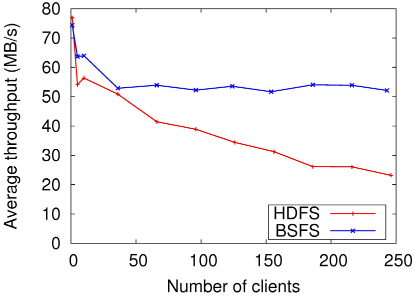
Design issues

- Implement Hadoop API
- Hierarchic namespace for BLOBs
- Data prefetching
- Affinity scheduling: exposing data location

Results: synthetic benchmarks

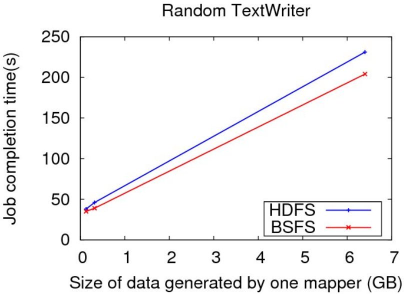


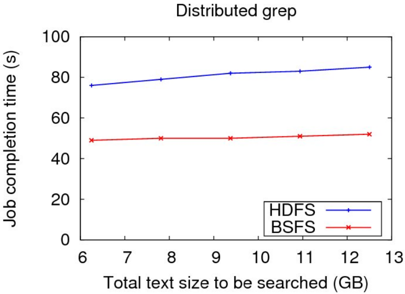
Single writer



Concurrent readers

Results: Real MapReduce applications





Write intensive

Read intensive

In short

- Improvement of 11%-30% over HDFS for real MapReduce applications
- Potential to leverage versioning in Hadoop
  - Further improve performance
  - New features

Virtual machine image storage for IaaS clouds

- On IaaS clouds users rent resources as VMs
- VM image customized with user application
- Two patterns:
  - Multi-deployment: instantiate many VMs from the same image
  - Multi-snapshotting: save state of VMs into independent images
- State-of-art: full pre-propagation, then copy images back to repository
- Our goal: reduce cost (execution time, storage space, network traffic)


VM image
VM instance
VM instance
VM instance
VM image
VM image
VM image

Proposal

Store image in a striped fashion

- Leverage BlobSeer to store each image as a BLOB

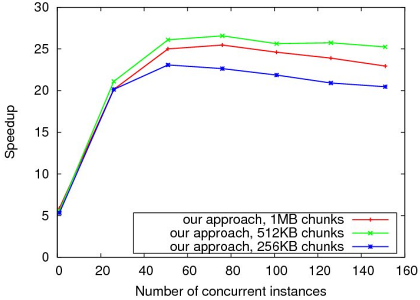
Mirror image contents locally

- Lazy scheme: only read from BLOB when needed
- Keep changes to image local
- Only the necessary parts are accessed

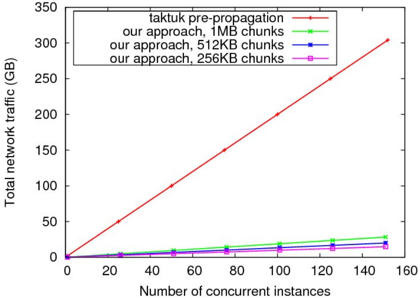
Consolidate local changes into an independent image

- CLONE BLOB, then WRITE local changes to BLOB
- Provides illusion of independent images
- Only differences are stored

Results: scalability of multi-deployment under concurrency

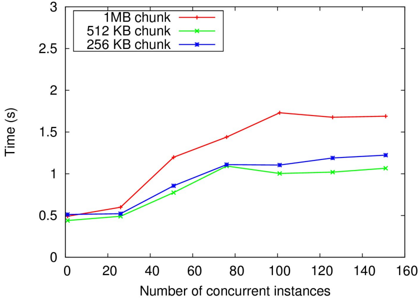


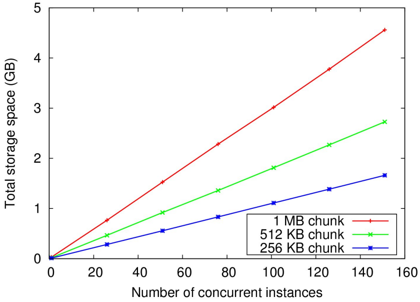
VM boot: runtime speedup vs. pre-propagation



VM boot: total network traffic vs. pre-propagation

Results: performance of multi-snapshotting





Avg. time to take snapshots

Total network traffic (storage space)

In short

- Large speedup and network traffic savings over pre-propagation for multi-deployment
- Efficient multi-snapshotting
- Portable approach: does not depend on hypervisor to manage diffs

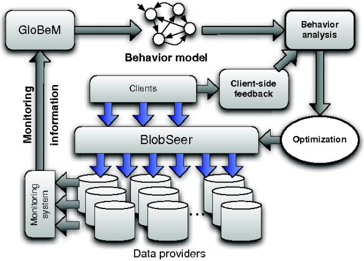
Quality-of-service enabled storage for cloud applications

- Storage for data-intensive applications deployed on IaaS clouds
- QoS impacted by several factors
  - Multiple customers share the same storage service
  - Hardware components prone to failures
  - Application access pattern
- We need:
  - High aggregated throughput under concurrency
  - Stable throughput for individual data accesses

Proposal: QoS improvement methodology

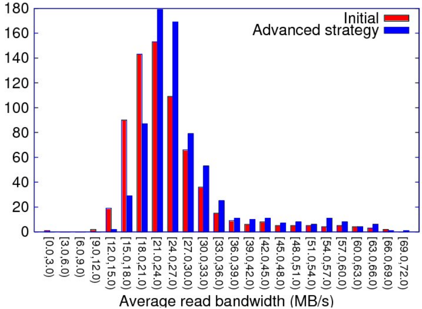
Methodology

1. Monitor storage service
2. Collect app feedback
3. Identify + classify behavior patterns using GloBeM
4. Prevent undesired patterns

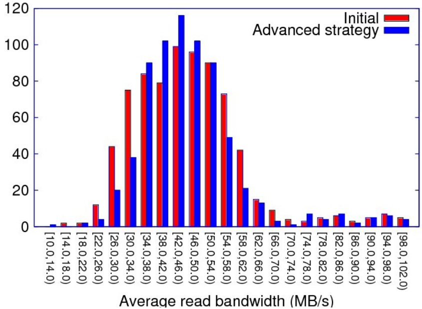


- Input: MapReduce access patterns + faults
- Applied methodology to find bottlenecks
- Output: Improved BlobSeer allocation strategy

Results: improvement of throughput stability



Clients separated from providers



Clients co-deployed with providers

In short

- General methodology to improve QoS for cloud storage
- Concretely:
    - Reduction in standard deviation for read throughput of up to 25%
    - Promising results for cloud providers to improve SLA for the same price