Emergent Properties in Modular Storage

A Study of Apple Desktop Applications, Facebook Messages, and Docker Containers

Tyler Harter
How are complex applications built? (for example, Facebook Messages)
We have many machines with many disks.

*How should we use them to store messages?*
One option: use machines and disks directly.
One option: use machines and disks directly. Very specialized, but very high development cost.
Use **HBase** for K/V logic
Use **HBase** for K/V logic
Use **HDFS** for replication
Use **HBase** for K/V logic
Use **HDFS** for replication
Use **Local FS** for allocation
Modules Divide Work

FB Messages (application logic)
  ↓
  HBase (database logic)
  ↓
  HDFS (replication logic)
  ↓
  Local FS (allocation logic)
  ↓
  Block Device (scheduling logic)
Modularity Enables Reuse

FB Messages → HBase

Hive → MapReduce

HDFS

Local FS → Block Device
Modularity Enables Reuse

- HBase
- MR
- HDFS
- HBase
- MR
- HDFS
- Local FS
- Block Device
How are complex applications built? (for example, Facebook Messages)
How are complex applications built? (for example, Facebook Messages)

**Answer:** *by gluing together existing components*
Conceptual Integrity

**Conceptual integrity** “dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds.”

~ Frederick Brooks, *The Mythical Man-Month*
Conceptual Integrity

Conceptual integrity “dictates that the design must proceed from one mind, or from a very small number of agreeing resonant minds.”

~ Frederick Brooks, *The Mythical Man-Month*

**Premise**: modern applications and storage systems are patched together and lack conceptual integrity.
Emergent Properties

Emergent Properties: “properties that are not evident in the individual components, but they show up when combining those components”

~ Saltzer and Kaashoek, Principles of Computer System Design

“they might also be called surprises”
Summarizing Modern Storage

Storage systems benefit from modular
• Modules divide work
• Modules enable reuse

But these systems lack conceptual integrity

Questions
• What are the storage needs of modern applications?
• What impact does modularity have on I/O patterns?
• How can we better modularize storage systems?
Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions
Many Types of Reuse

- Msgs
- Hive
- iTunes
- Pages
- RabbitMQ
- App
- Django
- Docker
- AUFS

- HBase
- MR
- HDFS
- Cocoa
- SQLite
- Docker

- Local FS
- Block Device
Many Types of Reuse

- Msgs → HBase
- Hive → MR
- HDFS
- iTunes → Pages
- Cocoa → SQLite
- RabbitMQ
- App → Django
- Docker
- AUFS
- Local FS
- Block Device

libraries
Many Types of Reuse

- Local FS
- AUFS
- HDFS
- Cocoa
- SQLite
- HBase
- Hive
- MR
- iTunes
- Pages
- RabbitMQ
- App
- Django
- Docker
- AUFS

Layers:
- HBase → MR → HDFS
- Cocoa → SQLite
- Many Types of Reuse: libraries, layers
Many Types of Reuse

libraries
layers

Msgs → Hive → MR → HBase → HDFS

iTunes → Pages

Cocoa → SQLite

RabbitMQ → App

Django

Docker

AUFS

Local FS

Block Device
This Dissertation

iLife and iWork study

iTunes  Pages

HBase  MR  HDFS

Cocoa  SQLite

Local FS

Block Device

Msgs  Hive

RabbitMQ

App  Docker  AUFS

Django
This Dissertation

- HDFS
- Hive
- iTunes
- Pages
- Cocoa
- SQLite
- R (RabbitMQ)
- Docker
- AUFS
- Django
- App
- Local FS
- Block Device
- Facebook Messages study
- iLife and iWork study

- HBase
- MR
-Msgs
This Dissertation

- iTunes
- Pages
- HBase
- MR
- HDFS
- Cocoa
- SQLite
- Local FS
- Block Device
- AUFS
- Docker
- RabbitMQ
- App
- Django
- startup study
- iLife and iWork study

- Facebook Messages study

- Facebook
- Messages
- study

- HDFS
- AUFS
- Docker
Publications

**SOSP ’11:** A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau.

**TOCS ’12:** A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications. Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau.

**FAST ’14:** Analysis of HDFS Under HBase: A Facebook Messages Case Study. Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

**;login ’14:** Analysis of HDFS Under HBase: A Facebook Messages Case Study. Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

**FAST ’16:** Slacker: Fast Distribution with Lazy Docker Containers. Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions
In 1974:

“No large ‘access method’ routines are required to insulate the programmer from the system calls; in fact, all user programs either call the system directly or use a small library program, only tens of instructions long…”

~ Ritchie and Thompson. The UNIX Time-Sharing System.
Modern Desktop Applications and Libraries

In the past, applications:
- Used the file-system API directly
- Performed simple tasks well
- Chained together for more complex actions
In the past, applications:
• Used the file-system API directly
• Performed simple tasks well
• Chained together for more complex actions

Today, we see:
• Applications are graphically rich, multifunctional monoliths
• “#include <Cocoa/Cocoa.h> reads 112,047 lines from 689 files” ~ Rob Pike ‘10
• They rely heavily on I/O libraries
Our Study

Measure 34 tasks from popular home-user applications

- **iLife suite (multimedia)**
  - iPhoto 8.1.1
  - iTunes 9.0.3
  - iMovie 8.0.5

- **iWork (like MS Office)**
  - Pages 4.0.3 (Word)
  - Numbers 2.0.3 (Excel)
  - Keynote 5.0.3 (PowerPoint)

Goal: understand I/O patterns and impact of libraries
Our Study

Measure 34 tasks from popular home-user applications

- iLife suite (multimedia)
  - iPhoto 8.1.1
  - iTunes 9.0.3
  - iMovie 8.0.5

- iWork (like MS Office)
  - Pages 4.0.3 (Word)
  - Numbers 2.0.3 (Excel)
  - Keynote 5.0.3 (PowerPoint)

This talk: look at one task from Pages in detail as case study
A Case Study: Saving a Document

Application: Pages 4.0.3
- From Apple’s iWork suite
- Document processor (like Microsoft Word)

One simple task (from user’s perspective):
1. Create a new document
2. Insert 15 JPEG images (each ~2.5MB)
3. Save to the Microsoft DOC format

Trace I/O System Calls
- Instrument with DTrace, record user-space stack traces
- Relatively little paging from mmap I/O
The image shows a graph with various categories on the y-axis: Other, Strings, Multimedia, SQLite, KV Store, and Documents. The x-axis represents seconds, ranging from 0 to 70. Each category has a line indicating the amount of small I/O and big I/O over time.

The graph highlights a period where there is a significant increase in big I/O activity, particularly in the Multimedia category, which is indicated by a red box. The data suggests that Multimedia operations are experiencing a peak in I/O activity, which could be due to large file operations or complex data transfers.

The chart provides insights into the system's performance, particularly in identifying bottlenecks and areas that require optimization, such as Multimedia operations for big I/O.
Case Study Observations

• **Auxiliary files dominate**
  - **Task’s purpose:** create 1 file; observed I/O: **385** files are touched
  - 218 KV store files + 2 SQLite files:
    - Personalized behavior (recently used lists, settings, etc)
  - 118 multimedia files:
    - Rich graphical experience
  - 25 Strings files:
    - Language localization
  - 17 Other files:
    - Auto-save file and others
Case Study Observations

• Auxiliary files dominate
• Multiple threads perform I/O
  • Interactive programs must avoid blocking
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- **Writes are often forced**
  - KV-store + SQLite durability
  - Auto-save file
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- **Renaming is popular**
  - Often used for key-value store
  - Makes updates atomic
Writing the DOC file
Writing the DOC file
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- **A file is not a file**
  - DOC format is modeled after a FAT file system
    - Multiple “sub-files”
    - Application manages space allocation
Writing the DOC file
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- **Sequential access is not sequential**
  - Multiple sequential runs in a complex file => random accesses
Writing the DOC file
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- Sequential access is not sequential

**Frameworks influence I/O**
- Example: update value in page function
- Cocoa, Carbon are a substantial part of application
Case Study Observations

- Auxiliary files dominate
- Multiple threads perform I/O
- Writes are often forced
- Renaming is popular
- A file is not a file
- Sequential access is not sequential
- Frameworks influence I/O

All findings are general trends across multiple tasks
(more details in dissertation)
Case Study Observations

• Auxiliary files dominate
• Multiple threads perform I/O
• Writes are often forced
• Renaming is popular
• A file is not a file
• Sequential access is not sequential
• Frameworks influence I/O

all findings are general trends across multiple tasks
(more details in dissertation)
Noted Effects of Modularity

Described in dissertation:

- Mismatch between .doc page size and STDIO block size
- Repeated read-copy-update to same page
- Open flags are meaningless (O_RDWR overused)
- Preallocation hints not meaningful
- Copy abstraction prevents combined use of source
- Coarse-grained exclusion make fine-grained locks useless
- Atomicity/durability required for unimportant data
Noted Effects of Modularity

Described in dissertation:

• Mismatch between .doc page size and STDIO block size
• Repeated read-copy-update to same page
• Open flags are meaningless (O_RDWR overused)
• Preallocation hints not meaningful
• Copy abstraction prevents combined use of source
• Coarse-grained exclusion make fine-grained locks useless
• Atomicity/durability required for unimportant data
Use of Fs equiv

Older studies

• Baker et al.: 16% of data flushed by app. request (1991)
• Vogels: “In 1.4% of file opens that had write operations posted to them, caching was disabled at open time. Of the files that were opened with write caching enabled, 4% actively controlled their caching by using the flush requests.” (1999)

Newer study

• Kim et al.: SQLite write traffic itself is quite random with plenty of synchronous overwrites … apps use the Android interfaces oblivious to performance. A particularly striking example is the heavy-handed management of application caches through SQLite.” (2012)
Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

▪ Cellphone texts
▪ Chats
▪ Emails
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Represents HBase over HDFS

- Common backend at Facebook and other companies
- Similar stack used at Google (BigTable over GFS)
Why Study Facebook Messages?

Represents an important type of application. Universal backend for:

- Cellphone texts
- Chats
- Emails

Represents HBase over HDFS

- Common backend at Facebook and other companies
- Similar stack used at Google (BigTable over GFS)

Represents layered storage
Methodology

Actual stack

Messages
HBase
HDFS
Local FS
Methodology

New tracing layer
- Hadoop Trace FS (HTFS)
- Collects request details
- Reads/writes, offsets, sizes
- Not contents

Trace results
- 9 shadow machines
- Production requests mirrored
- 8.3 days
- 71TB of HDFS I/O
Methodology

Actual stack
- Messages
- HBase
- HDFS
- Local FS

HDFS Traces

MapReduce Analysis Pipeline

Workload Analysis
Methodology

Actual stack
- Messages
- HBase
- HDFS
- Local FS

HDFS Traces

MapReduce Analysis Pipeline

Workload Analysis

Simulated stack
- HBase + HDFS
- Local Traces (inferred)

Local Storage

Simulation Results
Methodology

Actual stack

Messages
HBase
HDFS
Local FS

HDFS Traces
Methodology

Actual stack

Messages
HBase
HDFS
Local FS

Background: how does HBase use HDFS?
HBase’s HDFS Files

Four activities do HDFS I/O:

- MemTable
- LOG
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging

After many put’s, buffer fills
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
HBase’s HDFS Files

Four activities do HDFS I/O:
- Logging
- Flushing
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing

Files accumulate after many flushes

HBase Memory

HDFS Files

LOG
DATA
DATA
DATA
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads

get’s may check many files

```
HBase Memory

MemTable

HDFS Files

LOG
DATA
DATA
DATA
```
HBase’s HDFS Files

Four activities do HDFS I/O:

▪ Logging
▪ Flushing
▪ Foreground reads

get’s may check many files
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction

Baseline I/O:

- **Flushing and foreground reads** are always required
HBase’s HDFS Files

Four activities do HDFS I/O:

- Logging
- Flushing
- Foreground reads
- Compaction

Baseline I/O:

- **Flushing and foreground reads** are always required

HBase overheads:

- **Logging**: useful for crash recovery (not normal operation)
- **Compaction**: useful for performance (not correctness)
Facebook Messages Outline

Background

Workload Analysis
- I/O causes
- File size
- Sequentiality

Layer Integration
- Local compaction
- Combined logging

Discussion
Workload Analysis Questions

At each layer, what activities read or write?

How large are created files?

How sequential is I/O?
Workload Analysis Questions

At each layer, what activities read or write?

How large are created files?

How sequential is I/O?
Cross-layer R/W Ratios

Baseline HDFS I/O: 1% writes

reads
writes

I/O (TB)
Cross-layer R/W Ratios

Baseline HDFS I/O:
- 1% writes

All HDFS I/O:
- 21% writes

I/O (TB)
Cross-layer R/W Ratios

Baseline HDFS I/O: 1% writes

All HDFS I/O: compact 21%

Local FS: R1 45% replicas

I/O (TB)
Cross-layer R/W Ratios

Baseline HDFS I/O: 1% writes

All HDFS I/O: compact 21%

Local FS: R1 45%

Disk: cache misses 64%
Workload Analysis Conclusions

1. Layers amplify writes: 1% => 64%
   - Logging, compaction, and replication increase writes
   - Caching decreases reads
Workload Analysis Questions

At each layer, what activities read or write?

How large are created files?

How sequential is I/O?
50% of files are <750KB
90% of files are <6.3MB
Workload Analysis Conclusions

1. Layers amplify writes: 1% \(\Rightarrow\) 64%
2. Files are very small: 90% smaller than 6.3MB
Workload Analysis Questions

At each layer, what activities read or write?

How large are created files?

How sequential is I/O?
Reads: Run Size

Percent of read I/O

- 0%
- 25%
- 50%
- 75%
- 100%

Run Size:
- 8KB
- 16KB
- 32KB
- 64KB
- 128KB
- 256KB
- 512KB
- 1MB
- 2MB
- 4MB
- 8MB
- 16MB
- 32MB
- 64MB
- 128MB
- 256MB
- 512MB
- 1GB
- 2GB
- 4GB
- 8GB
Reads: Run Size

50% of runs (weighted by I/O) < 130KB
80% of files are <256KB
Workload Analysis Conclusions

① Layers amplify writes: 1% => 64%
② Files are very small: 90% smaller than 6.3MB
③ Fairly random I/O: 130KB median read run
Facebook Messages Outline

Background

Workload Analysis
- I/O causes
- File size
- Sequentiality

Layer Integration
- Local compaction
- Combined logging

Discussion
Software Architecture: Workload Implications

Writes are amplified

- 1% at HDFS (w/o overheads) to 64% at disk (30GB RAM)
- We should optimize writes

61% of writes are for compaction

36% of writes are for logging
Replication Overview

Machine 1
- HBase Worker
- HDFS Worker

Machine 2
- HBase Worker
- HDFS Worker

Machine 3
- HBase Worker
- HDFS Worker
Problem: Network I/O (red lines)
Solution: Ship Computation to Data

Machine 1
- HBase Worker
- HDFS Worker

Machine 2
- HBase Worker
- HDFS Worker

Machine 3
- HBase Worker
- HDFS Worker
In Our Case, do *Local Compaction*

- **Machine 1**: HBase Worker → HDFS Worker
- **Machine 2**: HBase Worker → HDFS Worker
- **Machine 3**: HBase Worker → HDFS Worker
In Our Case, do *Local Compaction*

- **Machine 1**
  - HBase Worker
  - HDFS Worker
  - compaction

- **Machine 2**
  - HBase Worker
  - HDFS Worker
  - compaction

- **Machine 3**
  - HBase Worker
  - HDFS Worker
  - compaction
Local Compaction

Normally 3.5TB of network I/O
Local Compaction

Normally 3.5TB of network I/O

Local comp: 62% reduction
Local Compaction

Normally 3.5TB of network I/O
Local comp: 62% reduction
Local Compaction

Normally 3.5TB of network I/O

Local comp: 62% reduction

Network I/O becomes disk I/O

- 9% overhead (30GB cache)

- Compaction reads are
  (a) usually misses,
  (b) pollute cache

- Disk I/O is much cheaper
Related Work: Salus

Wang *et al.* built Salus, an implementation of the HBase interface that replicates DB compute as well as storage.

- Side effect: compaction work is replicated, so Salus does local compaction.

Finding: “Salus often outperforms HBase, especially when disk bandwidth is plentiful compared to network bandwidth.”
Typical HDFS Worker Receives Logs from 3
Problem: Extra Seeks for Logging

Machine 1
- HBase Worker
- LOG
- HDFS Worker

Machine 2
- HBase Worker
- LOG
- HDFS Worker

Machine 3
- HBase Worker
- LOG
- HDFS Worker

Disks
Solution: Combine Logs (New HDFS API)
Combined Logging

- foreground
- compaction
- logging

Latency (ms)

Disks
Combined Logging

Log writes 6x faster (15 disks)
Combined Logging

Log writes 6x faster (15 disks)
Combined Logging

Log writes 6x faster (15 disks)

Compaction 12% faster

- Less competition with logs
Combined Logging

Log writes 6x faster (15 disks)
Compaction 12% faster
- Less competition with logs
Combined Logging

- Log writes 6x faster (15 disks)
- Compaction 12% faster
  - Less competition with logs
- Foreground reads 3% faster

Diagram:
- Latency (ms) on the y-axis
- Disks on the x-axis
- Lines indicating performance metrics for foreground, compaction, and logging operations.
Combined Logging

Log writes 6x faster (15 disks)

Compaction 12% faster
- Less competition with logs

Foreground reads 3% faster

Puts do not block currently
- Very useful if `put()`’s were to block until logs on disk
Facebook Messages Outline

Background

Workload Analysis
- I/O causes
- File size
- Sequentiality

Layer Integration
- Local compaction
- Combined logging

Discussion
Conclusion 1: New Workload on an Old Stack

Original GFS paper:
- “high sustained bandwidth is more important than low latency”
- “multi-GB files are the common case”

We find files are small and reads are random
- 50% of files <750KB
- 50% of read runs <130KB

Comparison to previous findings:
- Chen et al. found HDFS files to be 23 GB at 90th percentile
- We find HDFS files to be 6.3 MB at the 90th percentile
Conclusion 2: Layering is not Free

Layering “proved to be vital for the verification and logical soundness” of the THE operating system ~ Dijkstra

Layering is not free
  - Over half of network I/O for replication is unnecessary

Layers can amplify writes, multiplicatively
  - Logging overhead (10x) with replication (3x) => 30x write amp
Outline

Motivation: Modularity in Modern Storage

Overview: Types of Modularity

Library Study: Apple Desktop Applications

Layer Study: Facebook Messages

Microservice Study: Docker Containers

Slacker: a Lazy Docker Storage Driver

Conclusions
Container Popularity

docker  Rocket  Core OS  kubernetes

Google  facebook  twitter  amazon

heroku  spoon.net
What is a Container?

Goal: provide lightweight virtualization (compared to VMs)

Operating systems have long virtualized CPU and memory

But many resources have not been historically virtualized:
  • file system mounts
  • network
  • host names
  • IPC queues
  • process IDs
  • user IDs
What is a Container?

Goal: provide **lightweight virtualization** (compared to VMs)

Operating systems have long virtualized **CPU** and **memory**

But many resources have not been historically virtualized:
- file system mounts
- network
- host names
- IPC queues
- process IDs
- user IDs

New namespaces are collectively called “**containers**”
- lightweight, like virtual memory
- old idea rebranded (Plan 9 OS)
OS-Level Virtualization

Proc A

Proc B

CPU

RAM
OS-Level Virtualization

Proc A

scheduler
(CPU)

CPU

Proc B

RAM
OS-Level Virtualization

- Proc A
  - CPU
  - scheduler (CPU)

- Proc B
  - PT
  - namespace (memory)
  - RAM
OS-Level Virtualization

Proc A

Proc B

CPU

RAM

PT

PT

map

map

ports

80

80

80

100

200
Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?
Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

coarse if sandboxes are expensive
(e.g., virtual machines are used)
Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

fine if sandboxes are cheap
(e.g., containers are used)
Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

each microservice must be initialized first
Implications for Microservices

Decomposing applications is an old technique.

How fine grained should the components be?

file system provisioning is an interesting problem
Resource Initialization

- CPU core
- page
- /bin/…
- /usr/…

compute  memory  storage
Resource Initialization

OS

container

CPU core
compute (minimal init)

page
memory (zeroing)

/bin/…
storage (/usr/…)

(100’s of MBs)
Theory and Practice

**Theory**: containers are lightweight
  - just like starting a process!
Theory and Practice

**Theory**: containers are lightweight
- just like starting a process!

**Practice**: container startup is slow
- Large-scale cluster management at Google with Borg [1]
- *25 second* median startup
- 80% of time spent on package installation
- contention for disk a bottleneck
- this problem "*has received and continues to receive significant attention*"

[1] Large-scale cluster management at Google with Borg.
Theory and Practice

**Theory**: containers are lightweight
- just like starting a process!

**Practice**: container startup is slow
- Large-scale cluster management at Google with Borg [1]
- **25 second** median startup
- 80% of time spent on package installation
- contention for disk a bottleneck
- this problem “*has received and continues to receive significant attention*”

**Startup time matters**
- flash crowds
- load balance
- interactive development

[1] Large-scale cluster management at Google with Borg.  
Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

Analysis

• Data distribution across layers
• Access patterns
Docker Background

Deployment tool built on containers

An application is defined by a file-system image
- application binary
- shared libraries
- etc.

Version-control model
- **extend** images by committing additional files
- **deploy** applications by pushing/pulling images
Containers as Repos

**LAMP** stack example
- commit 1: Linux packages (e.g., Ubuntu)
- commit 2: Apache
- commit 3: MySQL
- commit 4: PHP

Docker “layer”
- commit
- container scratch space

Central registries
- Docker HUB
- private registries
Push, Pull, Run

registry

worker
worker
worker
Push, Pull, Run

registry

worker

worker

worker
Push, Pull, Run

registry

worker
worker
worker
Push, Pull, Run

registry

worker

worker

worker

pull

pull
Push, Pull, Run

registry

worker

worker

worker
Push, Pull, Run

registry

worker

worker

worker
need a new benchmark to measure Docker push, pull, and run operations.
Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

Analysis

- Data distribution across layers
- Access patterns
HelloBench

Goal: stress container startup

- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready
HelloBench

Goal: stress container startup

• including push/pull
• **57 container images** from Docker HUB
• run simple “hello world”-like task
• wait until it’s done/ready
HelloBench

Goal: stress container startup

- including push/pull
- 57 container images from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready
HelloBench

Goal: stress container startup
  • including push/pull
  • **57 container images** from Docker HUB
  • run simple “hello world”-like task
  • wait until it’s done/ready

Development cycle
  • distributed programming/testing
HelloBench

Goal: stress container startup
- including push/pull
- **57 container images** from Docker HUB
- run simple “hello world”-like task
- wait until it’s done/ready

Development cycle
- distributed programming/testing

Deployment cycle
- flash crowds, rebalance

![Deployment cycle diagram]
## Workload Categories

<table>
<thead>
<tr>
<th>Language</th>
<th>Linux Distro</th>
<th>Database</th>
<th>Web Server</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>clojure</td>
<td>alpine</td>
<td>cassandra</td>
<td>glassfish</td>
<td>drupal</td>
</tr>
<tr>
<td>gcc</td>
<td>busybox</td>
<td>crate</td>
<td>httpd</td>
<td>ghost</td>
</tr>
<tr>
<td>golang</td>
<td>centos</td>
<td>elasticsearch</td>
<td>jetty</td>
<td>hello-world</td>
</tr>
<tr>
<td>haskell</td>
<td>cirros</td>
<td>mariadb</td>
<td>nginx</td>
<td>jenkins</td>
</tr>
<tr>
<td>hylang</td>
<td>crux</td>
<td>mongo</td>
<td>php-zendserver</td>
<td>rabbitmq</td>
</tr>
<tr>
<td>java</td>
<td>debian</td>
<td>mysql</td>
<td>tomcat</td>
<td>registry</td>
</tr>
<tr>
<td>jruby</td>
<td>fedora</td>
<td>percona</td>
<td></td>
<td>rethinkdb</td>
</tr>
<tr>
<td>julia</td>
<td>mageia</td>
<td>postgres</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mono</td>
<td>opensuse</td>
<td>redis</td>
<td></td>
<td></td>
</tr>
<tr>
<td>perl</td>
<td>oraclelinux</td>
<td>rethinkdb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>php</td>
<td>ubuntu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pypy</td>
<td>ubuntu</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>python</td>
<td>ubuntu-debootstrap</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>r-base</td>
<td>ubuntu-upstart</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rakudo-star</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ruby</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>thrift</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Web Framework

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>django</td>
<td>iojs</td>
<td>node</td>
<td></td>
</tr>
<tr>
<td>iojs</td>
<td>node</td>
<td>rails</td>
<td></td>
</tr>
<tr>
<td>node</td>
<td>rails</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rails</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Docker Outline

Container and Microservice Background

Docker Background

HelloBench Workload

Analysis

- Data distribution across layers
- Access patterns
Analysis Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?
Analysis Questions

How is data distributed across Docker layers?

How much image data is needed for container startup?
HelloBench images

- **circle**: commit
- **red**: image
Image Data Depth

half of data is at depth 9+
Analysis Questions

How is data distributed across Docker layers?

• half of data is at depth 9+
• design implication: flatten layers at runtime

How much image data is needed for container startup?
Analysis Questions

How is data distributed across Docker layers?

• half of data is at depth 9+
• design implication: flatten layers at runtime

How much image data is needed for container startup?
Container Amplification

![Container Amplification Graph](image)

- Distro: 1.8 MB
- Database: 32 MB
- Language: 26 MB
- Web-Server: 51 MB
- Web-Framework: 35 MB
- Other: 48 MB
- All: 27 MB
Container Amplification

I/O and Size (MB)

- distro: 30x
- database: 3.5x
- language: 8.3x
- web-server: 3.6x
- web-framework: 7.2x
- other: 3.0x
- ALL: 5.5x

Categories: reads, compressed (network)
Container Amplification

only 6.4% of data needed during startup
Analysis Questions

How is data distributed across Docker layers?

• half of data is at depth 9+
• **design implication**: flatten layers at runtime

How much image data is needed for container startup?

• 6.4% of data is needed
• **design implication**: lazily fetch data
Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)
- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories

Operations
- push
- pull
- run
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ➔ directory in underlying FS
- root FS ➔ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer \( \rightarrow \) directory in underlying FS
- root FS \( \rightarrow \) union of layer directories

AUFS Driver

directories:

A  B  C

...
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)
- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\Rightarrow$ directory in underlying FS
- root FS $\Rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories

PULL

directories:

A  B  C

tar.gz
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\Rightarrow$ directory in underlying FS
- root FS $\Rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)
- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories

RUN

scratch dir:

AUFS Driver

A B C

X Y Z
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\Rightarrow$ directory in underlying FS
- root FS $\Rightarrow$ union of layer directories

RUN

AUFS Driver

A B C

X Y Z

read B

AUFS
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer ⇒ directory in underlying FS
- root FS ⇒ union of layer directories

RUN

AUFS Driver

read B

AUFS
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories

RUN

AUFS Driver

read X

AUFS

A B C

X Y Z
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer \rightarrow directory in underlying FS
- root FS \rightarrow union of layer directories
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories

RUN AUFS
**AUFS Storage Driver**

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories

**Diagram:**
- RUN
- AUFS Driver
  - A B C
  - X Y Z
- append Z
- AUFS
AUFS Storage Driver

Uses AUFS file system (Another Union FS)
- stores data in an underlying FS (e.g., ext4)
- layer $\rightarrow$ directory in underlying FS
- root FS $\rightarrow$ union of layer directories

RUN

\begin{center}
\begin{tikzpicture}
  \node (A) at (0,0) {AUFS Driver};
  \node (B) at (1,-2) {X Y Z};
  \node (C) at (1,-3) {A B C};
  \node (D) at (2,-4) {Z};

  \draw[->,thick] (A) -- (B);
  \draw[->,thick] (B) -- (C);
  \draw[->,thick] (C) -- (D);
  \draw[->,thick] (D) -- (A);

  \node (E) at (3,-5) {AUFS};
  \node (F) at (2,-5) {copy};
  \node (G) at (4,-5) {append Z};

\end{tikzpicture}
\end{center}
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer $\Rightarrow$ directory in underlying FS
- root FS $\Rightarrow$ union of layer directories

RUN

AUFS Driver

append Z
AUFS Storage Driver

Uses AUFS file system (Another Union FS)

- stores data in an underlying FS (e.g., ext4)
- layer \( \rightarrow \) directory in underlying FS
- root FS \( \rightarrow \) union of layer directories

RUN

AUFS Driver

append Z
AUFS Storage Driver

Uses AUFS file system (Another Union FS)
- stores data in an underlying FS (e.g., ext4)
- layer → directory in underlying FS
- root FS → union of layer directories
HelloBench with AUFS

![Bar chart showing time in seconds for different categories and actions: push, pull, and run. The categories include distro, db, language, web server, web fwk, other, and ALL.]
HelloBench with AUFS

76% of deployment cycle spent on pull
AUFS Problems

Deployment problem: lots of copying

- Caused by push+pull
- Compute costs: compression
- Network costs: transferring tar.gz files
- Storage I/O costs: installing packages
- Pull+run = 26 seconds

Execution problem: coarse-grained COW

- Iterate over directories on lookup
- Large copies for small writes
- more in dissertation
Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation
Slacker Driver

Goals

• make push+pull very fast
• create drop-in replacement; don’t change Docker framework itself

Design

• lazily pull image data (like Nicolae et al. do for VMs)
• utilize COW primitives of Tintri VMstore backend (block level)
Prefetch vs. Lazy Fetch

Docker

registry

images

containers

worker

Slacker

registry

images and containers

worker
Prefetch vs. Lazy Fetch

Docker

registry

images

containers

worker

Slacker

registry

images and containers

Tintri

worker

significant copying
- over network
- to/from disk

centralized storage
- easy sharing
Prefetch vs. Lazy Fetch

Docker

registry

images

containers

worker

Slacker

registry

images and containers

worker
Prefetch vs. Lazy Fetch

Slacker

registry

container

ext4

loopback

NFS File
Prefetch vs. Lazy Fetch

Slacker

registry

container
ext4
loopback

NFS File
Prefetch vs. Lazy Fetch

Slacker

registry

container
ext4
loopback

NFS File

VMstore abstractions...
VMstore Abstractions

Copy-on-Write

- VMstore provides `snapshot()` and `clone()`

`snapshot(nfs_path)`
- create read-only copy of NFS file
- return snapshot ID

`clone(snapshot_id)`
- create r/w NFS file from snapshot

Slacker Usage

- NFS files → container storage
- snapshots → image storage
- `clone()` → provision container from image
- `snapshot()` → create image from container
Lazy Allocation

worker A

container

NFS file

Tintri VMstore
Lazy Allocation

Worker A: push
Lazy Allocation

Worker A: push
Lazy Allocation

Worker A: push
Lazy Allocation

Worker A: push
Lazy Allocation

Worker A: push

Tintri VMstore

worker A

registry

NFS file

snap N

img
Lazy Allocation

**Note**: registry is only a name server. Maps layer metadata $\Rightarrow$ snapshot ID

```
worker A ─registry
          ↘
          └── NFS file ➔ snap N
```
Lazy Allocation

worker A

registry

NFS file

snap N

Tintri VMstore
Lazy Allocation

Worker B: pull and run
Lazy Allocation

Worker B: pull and run
Lazy Allocation

Worker B: pull and run
Lazy Allocation

Worker B: pull and run
Lazy Allocation

Worker B: pull and run
Lazy Allocation

Worker B: pull and run
Indirection Discussion

File namespace level
- flatten layers
- if B is child of A, then “copy” A to B to start. Don’t make B empty

Block level
- do COW+dedup beneath NFS files, inside VMstore
Indirection Discussion

File namespace level

- flatten layers
- if B is child of A, then “copy” A to B to start. Don’t make B empty

Block level

- do COW+dedup beneath NFS files, inside VMstore

![Diagram showing namespace, copy-on-write, and block structures for AUFS and Slacker filesystems.]
Challenge: Framework Assumptions

Assumed Layout

```
| A | B | C | D |
```

Actual Layout

```
| A | B | C | D |
```

Layers
Challenge: Framework Assumptions

**Assumed Layout**

- A
- B
- C
- D

**Actual Layout**

- A
- B
- C
- D
  - pull

- A
  - pull

Layers
Challenge: Framework Assumptions

Strategy: **lazy cloning**. Don’t clone non-top layers until Docker tries to mount them.

**Assumed Layout**

```
A
B
C
D
```

**Actual Layout**

```
A
B
C
D
```

Layers

```
A
B
C
D
```
Slacker Outline

AUFS Storage Driver Background

Slacker Design

Evaluation
Questions

What are deployment and development speedups?

How is long-term performance?
Questions

What are deployment and development speedups?

How is long-term performance?
HelloBench Performance

development: push+pull+run
deployment: pull+run

Percent of Images

Slacker Speedup

0x  20x  40x  60x  80x  100x
Questions

What are deployment and development speedups?
• 5x and 20x faster respectively (median speedup)

How is long-term performance?
Questions

What are deployment and development speedups?
  • 5x and 20x faster respectively (median speedup)

How is long-term performance?
Server Benchmarks

Databases and web servers
- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment
- measure throughput (after startup)
- run 5 minutes
Server Benchmarks

Databases and web servers
- PostgreSQL
- Redis
- Apache web server (static)
- io.js Javascript server (dynamic)

Experiment
- measure throughput (after startup)
- run 5 minutes

Result: Slacker is always at least as fast as AUFS
Questions

What are deployment and development speedups?
  • 5x and 20x faster respectively (median speedup)

How is long-term performance?
  • there is no long-term penalty for being lazy
Slacker Conclusion

Containers are inherently lightweight
  - but existing frameworks are not

COW between workers is necessary for fast startup
  - use shared storage
  - utilize VMstore snapshot and clone

Slacker driver
  - 5x deployment speedup
  - 20x development speedup
Outline

**Motivation:** Modularity in Modern Storage

**Overview:** Types of Modularity

**Library Study:** Apple Desktop Applications

**Layer Study:** Facebook Messages

**Microservice Study:** Docker Containers

**Slacker:** a Lazy Docker Storage Driver

**Conclusions**
Modularity Often Causes Unnecessary I/O

Measurement exposed undesirable emergent properties

**Libraries** cause iBench applications to excessively flush

**Layers** cause Facebook Messages to waste network I/O

**Microservice** provisioning unnecessary copying
Layers Mask Costs

Apple desktop
• Key/value layer causes excessive fsync/rename
• SQLite use caused excessive fine-grained locking, rendered unnecessary by higher-level exclusion

Facebook Messages
• composition of layers amplifies writes from 1% to 64% of total I/O

Docker containers
• AUFS access surprisingly expensive to deep data
Simple Integration Surprisingly Useful

Measurement-driven optimizations are surprisingly effective at mitigating the cost of modularity

Local compaction
  • reduces network I/O by \( 2.7x \)

Combined logging
  • reduces log latency by \( 6x \)

Lazy propagation
  • reduces container startup latency by \( 5x \)
Thank you!