Your Storage is Broken

Lessons from Studying Databases and Key-Value Stores

Remzi H. Arpaci-Dusseau
Andrea C. Arpaci-Dusseau
Many Students
University of Wisconsin-Madison

Major Problem for
a Storage System:
Complexity

Complexity is Everything

Internal complexity:

Each system alone is complex

- Local file system has ~100k LOC
- Similar complexity in distributed FS, firmware, etc.
- What does system actually do?

Cross-system Complexity:

Connecting large systems multiplies complexity

- Deceptive APIs, hard-to-verify guarantees

An Example

Goal:
Update a Local File
(and tolerate crashes)

Should be simple, right?

Application on Local File System

Initial state of file `/f`:

`/f -> “a bar”`

Application on Local File System

Initial state of file `/f`:

`/f -> "a bar"` (pretend each char is block)

Application on Local File System

Initial state of file **`/f`**:

`/f -> “a bar”` (pretend each char is block)

Protocol:

`pwrite(file=/f, offset=2, data=“foo”)`

Application on Local File System

Initial state of file **`/f`**:

`/f -> "a bar"` (pretend each char is block)

Protocol:

`pwrite(file=/f, offset=2, data="foo")`

Final state of file:

`/f -> "a foo"`

What Is Atomic?

But `pwrite()` isn't atomic!

- Many intermediate states are possible

```
"a boo"
"a far"
"a for"
"a bor"
```

etc.

Use Logging!

Application protocol

- Create log file
- Make copy of old data in log
- Modify file with new data
- Delete log file

If crash occurs, recover old data from log

Update Protocol #2

```
create(/log)
write(/log, “2,3,bar”)

pwrite(/f, 2, “foo”)

unlink(/log)
```

Works on Linux ext3 (journal=data)

Update Protocol #2

```
create(/log)
write(/log, "2,3,bar")

pwrite(/f, 2, "foo")

unlink(/log)
```

Why?
Writes may
be reordered!

Works on Linux ext3 (journal=data)
Doesn't work in ext3 (journal=ordered)

Update Protocol #3

```
create(/log)
write(/log, “2,3,bar”)
fsync(/log)
pwrite(/f, 2, “foo”)
fsync(/f)
unlink(/log)
```

Why?
inode+data write
are not atomic
(may find garbage
at end of log)

Works in ext3 (data, ordered)
Doesn’t work in ext3 (writeback)!

Update Protocol #4

```
create(/log)
write(/log, “2,3,checksum,bar”)
fsync(/log)
pwrite(/f, 2, “foo”)
fsync(/f)
unlink(/log)
```

Works in ext3 (data, ordered, writeback)!

Update Protocol #4

```
create(/log)
write(/log, "2,3,checksum,bar")
fsync(/log)
pwrite(/f, 2, "foo")
fsync(/f)
unlink(/log)
```

Why?
directory entry
for `/log` not
made durable

Works in ext3 (data, ordered, writeback)!
Well, except dir fsync() is missing …

Update Protocol #5

```
create(/log)
write(/log, “2,3,checksum,bar”)
fsync(/log)
fsync(/)
pwrite(/f, 2, “foo”)
fsync(/f)
unlink(/log)
```

Update Protocol #5

```
create(/log)
write(/log, “2,3,checksum,bar”)
fsync(/log)
fsync(/)
pwrite(/f, 2, “foo”)
fsync(/f)
unlink(/log)
```

Each file system may be different;
Each application protocol is interesting

This Talk

Tool 1: **BOB** to study of FS **persistence properties**

- Automated tool to determine properties
- Surprise: **File systems vary widely**

Tool 2: **ALICE** to study application **update protocols**

- Automated tool to find **crash vulnerabilities**
- Surprise: **Even battle-tested apps are buggy**

Systems #1: **Optimistic File System** (OptFS)

- Achieves performance and correctness for many apps

Concluding thoughts and "one more thing"

File System
Persistence Properties

Background: File Systems

The File System API: Simple, right?

- Just open(), read(), write(), close(), etc. ?

But, there are some subtleties

Examples

- Does rename() complete in all-or-none fashion?
- Does write(A) reach disk before write(B)?
- How to ensure newly-created file is persisted?

Persistence Properties

Persistence properties of a file system:
Which post-crash on-disk states are possible?

Assertion:

- Different file systems have **different** properties (making life difficult for layer above)

Two Broad Properties

Atomicity

- Does update happen all at once, or in pieces?
- Example: write(block), rename(), etc.

Ordering

- Does A before B in **program order** imply A before B in **persisted order?**
- e.g., write(), write() ordering maintained?

Block Order Breaker (BOB)

How to discover properties of file system?

New tool: **Block Order Breaker (BOB)**

- **Run workloads:** Input for file system
- **Trace block I/O:** Monitor writes to disk
- **Emulate thousands of possible crashes:** Create possible on-disk states by applying subsets/reordering of I/Os to file-system image
- **Determine outcomes:** Examine image after FS recovery; find where properties don't hold

BOB Results

All file system operations grouped into …

- File overwrite
- File append
- Directory operations (rename, create, link, etc.)

Vary **size** of operations where needed

- Single sector
- Single block
- Multiple blocks

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

Results

		ext2	ext2 sync	ext3 wb	ext3 ord	ext3 data	ext4 wb	ext4 ord	ext4 nda	ext4 data	btrfs	xfs	xfs ws
Atomicity	1-sector overwrite												
	1-sector append	X		X			X						
	1-block overwrite	X	X	X	X		X	X	X			X	X
	1-block append	X	X	X			X						
	N-block write/append	X	X	X	X	X	X	X	X	X	X	X	X
	N-block prefix append	X	X	X			X						
	Directory operation	X	X										
Ordering	Overwrite - Any	X		X	X		X	X	X			X	X
	[Append, rename] - Any	X		X			X						
	O_TRUNC append - Any	X		X			X						
	Append - Append	X		X			X						
	Append - Any op (samefile)	X		X			X	X			X	X	
	Dir op - Any op	X									X		

BOB Summary

Persistence properties vary widely

- Different file system means different behavior
- How to write portable, correct applications?

Can sometimes rely on atomicity, order

- But have to be careful

BOB is a first step towards understanding

Question: **What does it mean for applications?**

Application
Crash Vulnerabilities

ALICE Goals

Each application has an **update protocol:**
the series of system calls it invokes to update persistent file-system state

Goals

- Determine update protocol
- Given persistence properties of file system, determine correctness of update protocol

Example

// a data logging protocol from BDB

```
creat(log)
trunc(log)
append(log)
```

What's missing?

- Truncate must be atomic
- Need fdatasync() at end

Can we discover these things automatically?

Application-Level Crash Explorer (ALICE)

How ALICE works

- Run workload
- Obtain **system-call trace**
- Transform into **micro-operations** (i.e., minimal atomic updates of file system state)
- **Emulate thousands of crashes:** Apply **persistence model** (how FS behaves) to determine possible post-crash states
- **Run workload checker** to determine if data store is consistent and has correct contents

From syscalls to µops

System calls: Large variety

- write(), pwrite(), writev(), pwritev(), mmap'd writes
- Also: creat(), link(), unlink(), rename(), etc.

Map into simple set of **micro-operations:**

- write-block atomic write of given size
- change-file-size atomic inc/dec of file size
- create-dentry atomic creation of dir entry
- delete-dentry atomic deletion of dir entry

Persistence Model

Persistence model

- Determines how file system restricts **atomicity** and **ordering** of operations
- Example (atomicity):
  Write(4KB) turns into 8 512-byte atomic writes;
  emulate all possible crashed states

Focus: Minimal abstract file system

- Provides weakest file-system guarantees possible
- Uncover **application correctness issues**

Also can model other modern file systems

- ext3, ext4, btrfs, etc.

Workload Checker

After generating post-crash states,
must determine if something “bad” happened

Workload checker serves this role

One needed per application
- Must run app-specific recovery
- Must determine health of app data store

Can be somewhat complex:
100s of lines of code per application

Applications

KV Stores

- LevelDB, GDBM, LMDB

Relational DBs

- SQLite, PostgresQL, HSQLDB

Version Control Systems

- Git, Mercurial

Distributed Systems

- HDFS, ZooKeeper

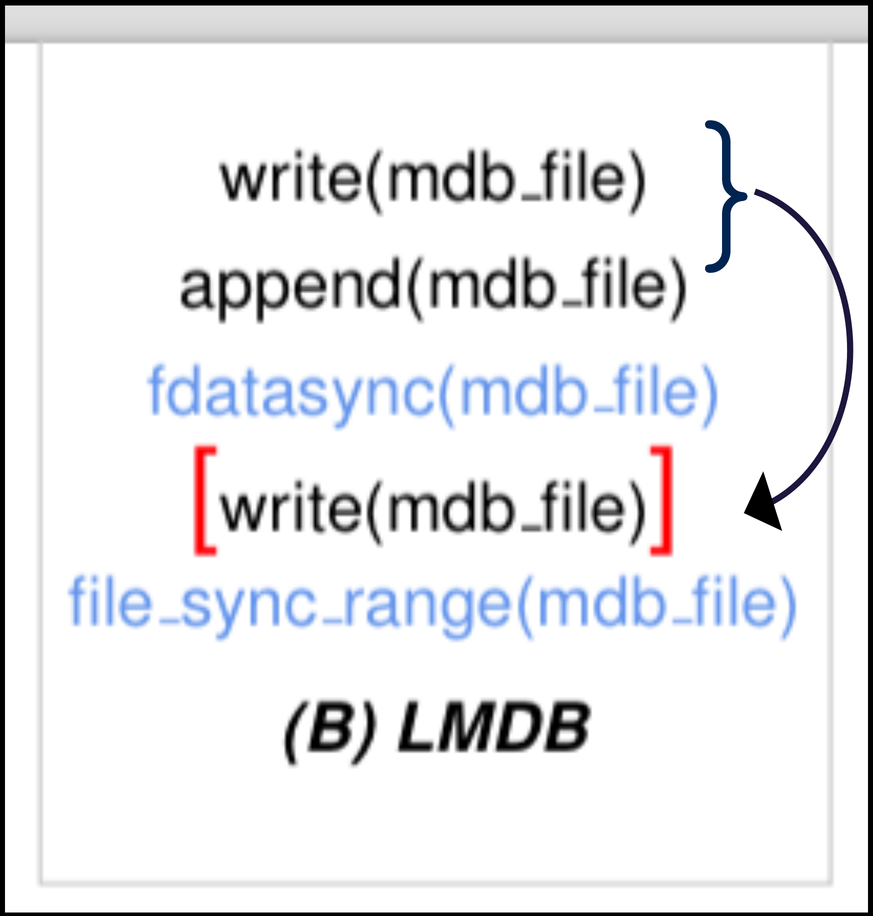
Virtualization Software

- VMWare Player

ALICE: Results

Protocol Diagrams

Output of ALICE: **Protocol Diagrams**



Blue:
sync() operation

[Red brackets]
Required atomicity

Arrows:
Required ordering

creat(x.ldb)
N x append(x.ldb)
fdatasync(x.ldb)
? x { creat(new.log)
creat(mani-new) }
[N x append(mani-new)]
fsync(parent-dir)
fdatasync(mani-new)
? x { creat(tmp)
append(tmp)
fdatasync(tmp)
[rename(tmp, current)]
unlink(mani-old) }
unlink(old.log)

(A)(i) LevelDB compaction

creat(new.log)
[N x append(new.log)]
? x fdatasync(new.log)
stdout(done)

(A)(ii) LevelDB insert

0 mkdir(o/x)
1 creat(o/x/tmp_y)
2 N x append(o/x/tmp_y)
3 fsync(o/x/tmp_y)
4 link(o/x/tmp_y, o/x/y)
5 unlink(o/x/tmp_y)

(G)(i) Git store object

creat(index.lock)
N x (i) store object
append(index.lock)
[rename(index.lock, index)]
stdout(finished add)
N x (i) store object
creat(branch.lock)
append(branch.lock)
append(branch.lock)
append(logs/branch)
append(logs/HEAD)
rename(branch.lock, x/branch)
stdout(finished commit)

(i)0,(i)4

(i)0,(i)4

(i)0,(i)4

(G)(ii) Git add commit

0 creat(tmp)
1 append(tmp)
2 fsync(tmp)
3 [unlink(props)]
4 rename(tmp, props)

(D)(i) HSQLDB update props

[append(log)]
N x fsync(log)
creat(stmp)
append(stmp)
fsync(stmp)
(i) update props
unlink(log)
unlink(script)
[rename(stmp, script)]
(i) update props

i(4)*

i(4)*

(i)4

(i)3

(i)3

(D)(ii) HSQLDB shutdown

Vulnerabilities Found

	Atomicity	Ordering	Durability
LevelDB 1.1	1	4	3
LevelDB 1.15	1	3	
LMDB	1		
GDBM	1	2	2
HSQLDB	1	6	3
SQLite			1
PostgreSQL	1		
Git	1	7	1
Mercurial	5	6	2
VMWare		1	
HDFS		2	
ZooKeeper	1	1	2

Some Highlights

Examples
- Surprising reliance on atomicity across sys calls
- Many cases where ordering assumed
- Append atomicity needed (garbage not tolerated)
- Small-write atomicity sometimes needed
- `fsync()` assumed to create file name durably

Some serious consequences too
- Data loss and silent corruption in worst cases
- "Cannot open database" in others

Reaction from Devs

Us: *There seems to be a bug in your app*

Reaction

Us: *There seems to be a bug in your app*
Developer: *That's not POSIX!*

Us: *Some applications assume garbage can never end up in the end of a file after append; we show it can, given modern file systems.*
Professor: *In* ***my*** *class, students who allow garbage to reside in a file will receive a failing grade.*

On Real File Systems?

Results: On abstract minimal FS

- Similar to ext2 (few guarantees)
- **~60 vulnerabilities** found across all apps

How do applications do on real file systems?

	Vulnerabilities
ext3 (writeback)	19
ext3 (ordered)	11
ext3 (data)	9
ext4 (ordered)	13
btrfs	31

ALICE Summary

Correctness issues found in all applications

- Some more problematic than others

All types of issues found

- Atomicity, ordering, and durability problems
- Many vulnerabilities manifest on **current FSes**

Beyond this work

- ALICE for distributed systems: **PACE**
- Crash behavior of scalable distributed file systems, key-value stores, and databases

Part 2: Build

Performance AND Correctness

Ordering Required

File Systems need to **order** writes

- Journaling (ext3, ext4, XFS, NTFS)
- Copy-on-write (ZFS, btrfs, WAFL)
- Soft updates (BSD)

Applications need to **order** writes too

- Using fsync()

How To Order Writes Within File System?

Modern devices (e.g., disks) have caches

Writes generally issued asynchronously

- Persistence order ≠ issue order

Use **cache-flush command** to ensure ordering

- To guarantee blkwrite(A) before blkwrite(B), issue blkwrite(A), **flush,** blkwrite(B)

How To Order Writes Within Application?

As seen, ordering required by many application-level update protocols

Use **fsync()** to ensure ordering

- To guarantee write(A) before write(B), issue write(A), **fsync(),** write(B)

The Problem

File systems **conflate ordering and durability**

- Internal to file system with cache flush
- External to applications with fsync()

Systems and applications are either…

- **Too slow:** Call fsync() or flush too often
- **Incorrect:** Don't call fsync() or flush enough

Optimistic File System

OptFS **separates ordering and durability**

- Internally: Avoids cache flushes in journaling protocol
- Externally: Provides **`osync()`** to applications (avoid force-to-disk to guarantee ordering)

Both file system and applications benefit

- Higher performance (~10x)
- Delivers **prefix consistency:** older (but consistent) data available after crash

Details: How Journaling Works

Example: File Append

Workload:

Application appends block to single file

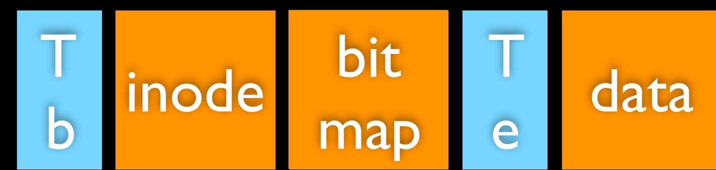
Must update file-system structures **atomically**

- Bitmap   Mark new block as allocated
- Inode   Point to new block
- Data block   Contain data of append

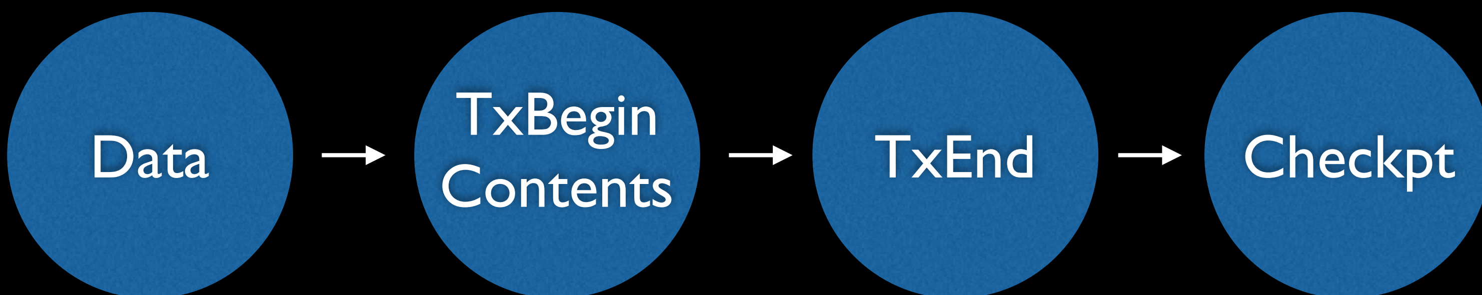
Ordered Journaling

Protocol: Write...

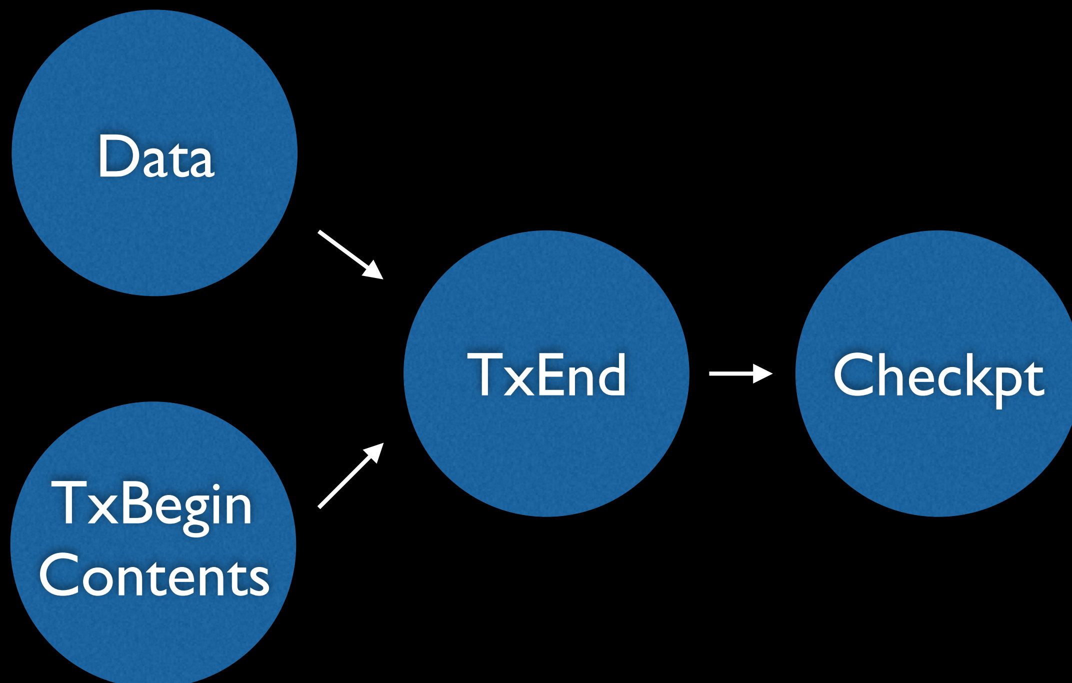
- Data
- Transaction Begin + Metadata
- Transaction **Commit**
- **Checkpoint** inode, bitmap



Ordering Required



Ordering (Precise)



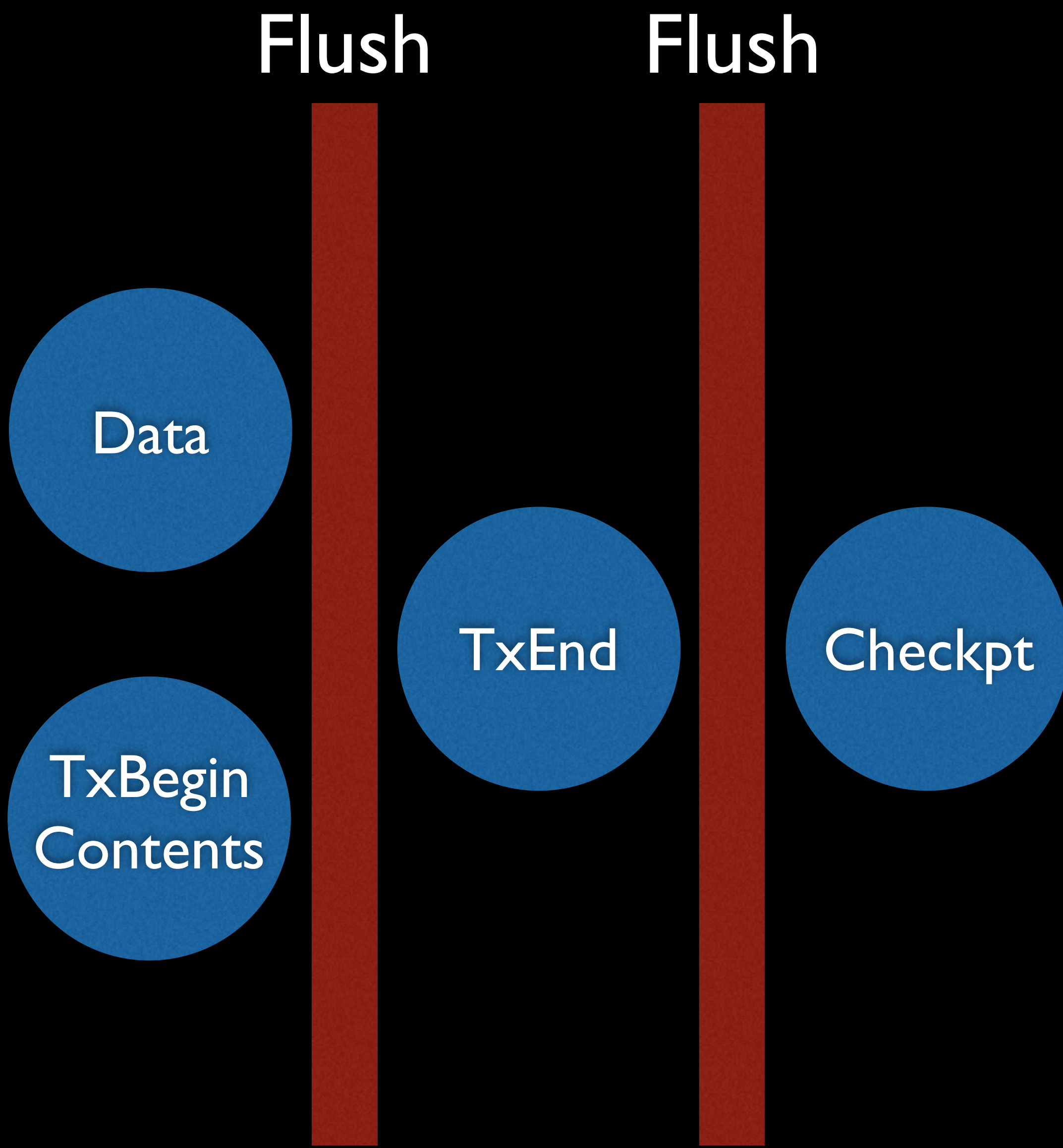
Ensuring Correctness

As stated before, **cache flushes** used to ensure ordering in protocol

Flush ensures all dirty data in disk cache is persisted before indicating completion

What is the cost of frequent flushing?

Graphically (w/ Flushes)



Durability: Expensive

Workload

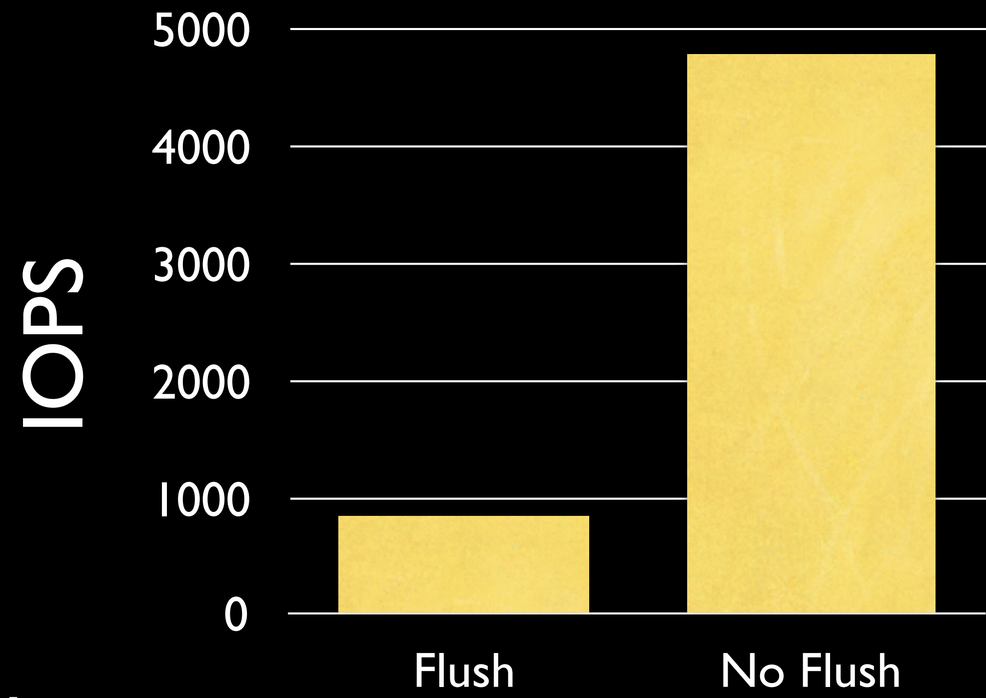
- varmail

System

- Linux ext4

Varying

- Cache flush on/off



Result: Must avoid flushing

- How to do so and realize journaling?

Optimistic Crash Consistency

Optimistic Approach

Realize: Most of the time, system doesn't crash

Use **checksums** (+other techniques) to avoid flushes

- Assumes slightly different disk interface
- Other techniques needed too - but not discussed (delayed block reuse and selective data journaling)

New file system interfaces too: no `fsync()`

- `osync()` : just for ordering
- `dsync()` : if you must have durability

Prefix Consistency

System call sequence

```
write(fd, A);
osync(fd);
write(fd, B);
```

Prefix consistency - disk could contain:

- Nothing
- Just `A`
- `A` and `B`
- … but **never** `B` without `A`

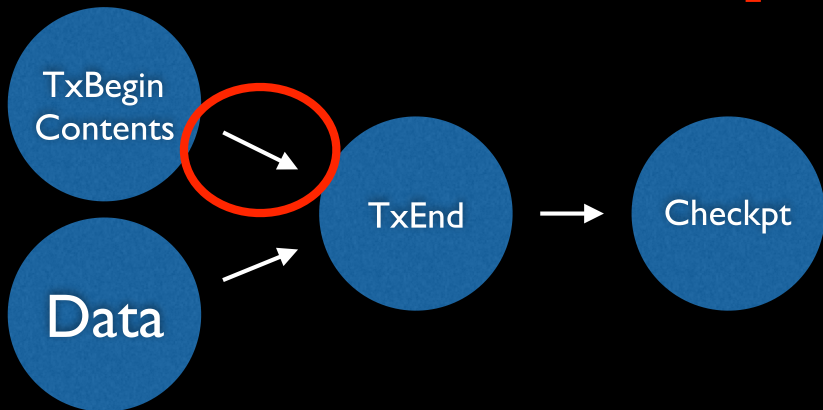
Classic `fsync()` guarantees same thing…

- … but immediately forces first write to disk (slow)

Building OptFS

Transaction Checksums

Key idea: **Use checksums to replace ordering**

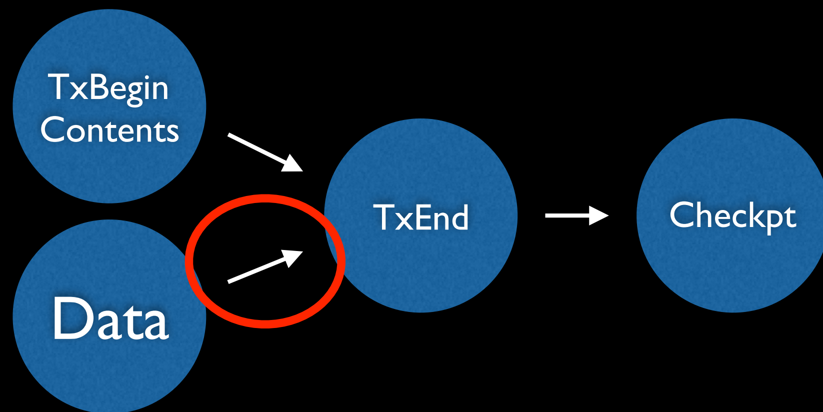


How to avoid ordering? **Transactional checksums**

- Compute checksum over journal
- Can write journal metadata and commit together
- Upon crash: redo iff checksum matches contents
- Idea from IRON File Systems [SOSP ’05] (later deployed in Linux ext4)

Data Checksums

Another problem: **Data blocks**



Solution: **Data checksums**

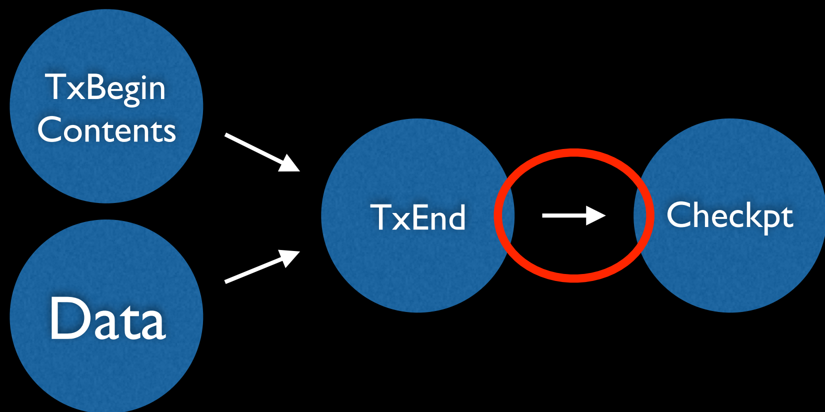
- Add checksums of pointed-to data in log

When used?

- If no crash: no problem (common case)
- If crash: checksum mismatch means discard data

One More Problem

Must separate journaling from checkpointing



How to know when logging is complete?

- Goal: Trying to avoid expensive cache flush

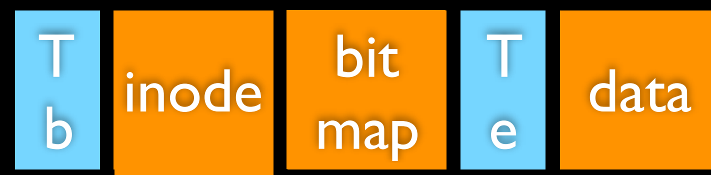
New: **Async Durability Notification (ADN)**

- After writes are persisted, OS notified by drive
- Simple way to know that protocol can proceed

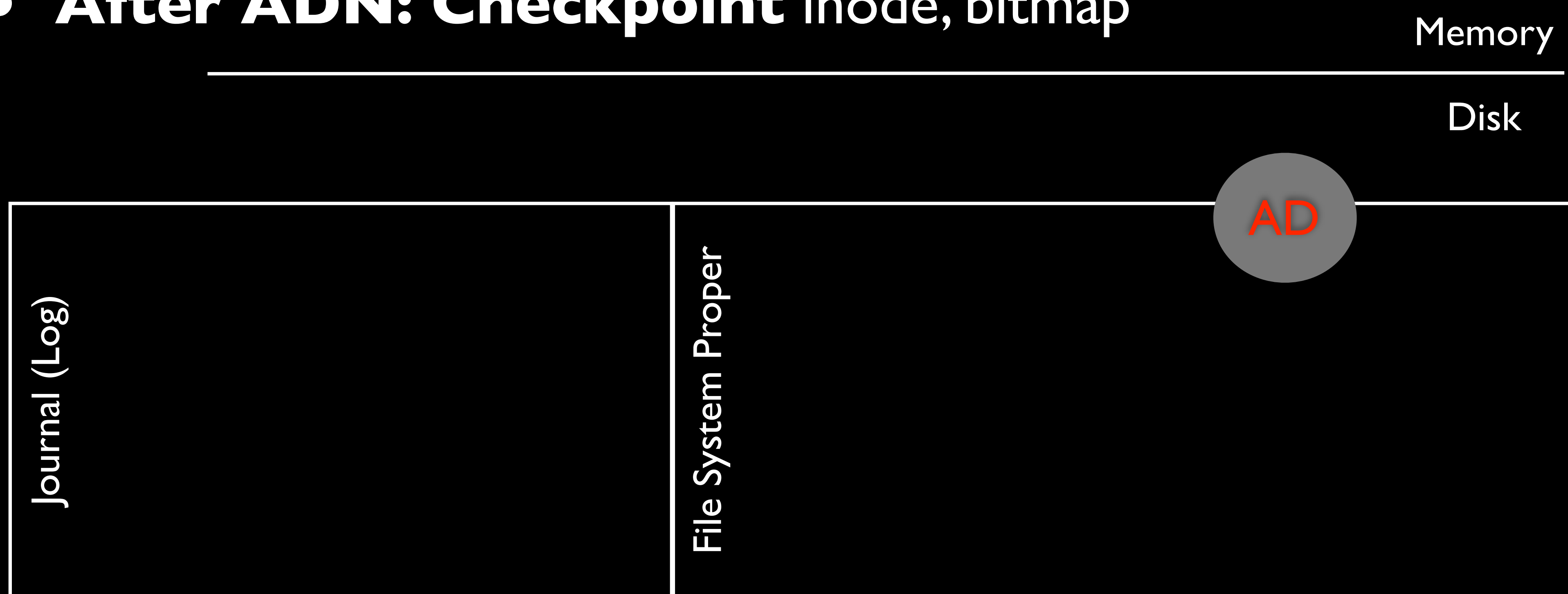
Optimistic Journaling

Protocol: Write...

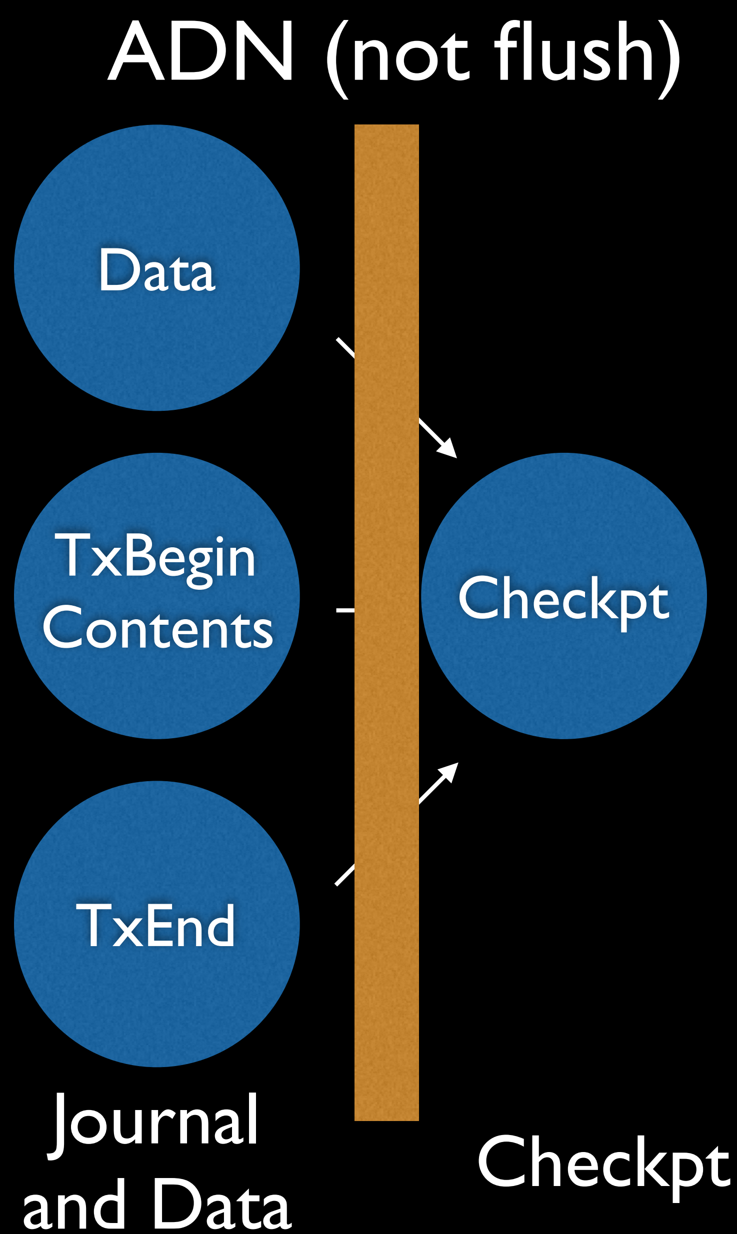
- Data
- Transaction Begin + Metadata



- Transaction **Commit**
- **After ADN: Checkpoint** inode, bitmap



Optimistic Journaling



OptFS Streamlines Multiple Transactions

Logging across multiple transactions happens first

- Journal + Data of T1
- Journal + Data of T2
- Journal + Data of T3
- …

Only much later does checkpointing take place

- Checkpoint of T1
- Checkpoint of T2
- Checkpoint of T3
- …

Optimistic Analysis

Empirical Evaluation

Workload

- SQLite table updates

Use new primitive **osync()**

- `write(A), osync(), write(B)` and similar constructs used where possible

Evaluate

- Does OptFS provide **prefix consistency?**
- How much does OptFS improve performance?

SQLite Analysis

	ext4 (fast/risky)	ext4 (slow/safe)	OptFS
Crashpoints	100	100	100
Inconsistent	73	0	0
$\text{Consistent}_{[old]}$	8	50	76
$\text{Consistent}_{[new]}$	19	50	24
Time/op	~15 ms	~150 ms	~15 ms

OptFS: **Fast and crash consistent**

- **10x faster** than slow/safe ext4
- Prefix consistency: **Always consistent** (but old?)

OptFS Summary

OptFS: **Separate durability from ordering**

- Internally: Avoid flushes
- Externally: Allow via osync()

Result: Performance and consistency

- Faster than classic ext4
- Does so while providing **prefix consistency**
- Idea already in use elsewhere (e.g., Blizzard)

Concluding Thoughts

and one more thing

Conclusions

Lack of clarity around crash consistency

First steps: Tools to analyze

- **BOB:** Find **persistence properties** of file systems
- **ALICE:** Find **update protocols + vulnerabilities**
- Current: distributed version of ALICE
- Others following up: Washington, Columbia, MIT

OptFS: Don't conflate durability and ordering

- Achieve high performance and prefix consistency
- Current: StreamFS to guarantee order of all writes

But more is needed

- Tools, systems, standards, new devices
- Study across layers: apps to storage to devices

Acknowledgements

Research “led” by Professors

- Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau

Real work (in this talk) done by

- Vijay Chidambaran, (@Texas), Thanu S. Pillai (Google), Ram Alagappan, Aishwarya Ganesan, Samer Al-Kiswany (@ Waterloo)

Papers: “Iron File Systems” (SOSP ’05), “Optimistic Crash Consistency” (SOSP ’13), “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications” (OSDI ’14)

And One More Thing…

The Case for FOBs

Lots of talk about free online classes **(MOOCs)**

- But what about materials?
- Books can be expensive …

Our goal: **Free Online textBooks (FOBs)**

Many reasons to make books freely available

- Share your knowledge with largest group possible
- Material easily found (Google) and linked to (Wikipedia)
- Self-printing sites a reality (lulu.com)
- Books usually written for reasons other than $$$

FOB #1: OSTEP

Operating Systems: Three Easy Pieces

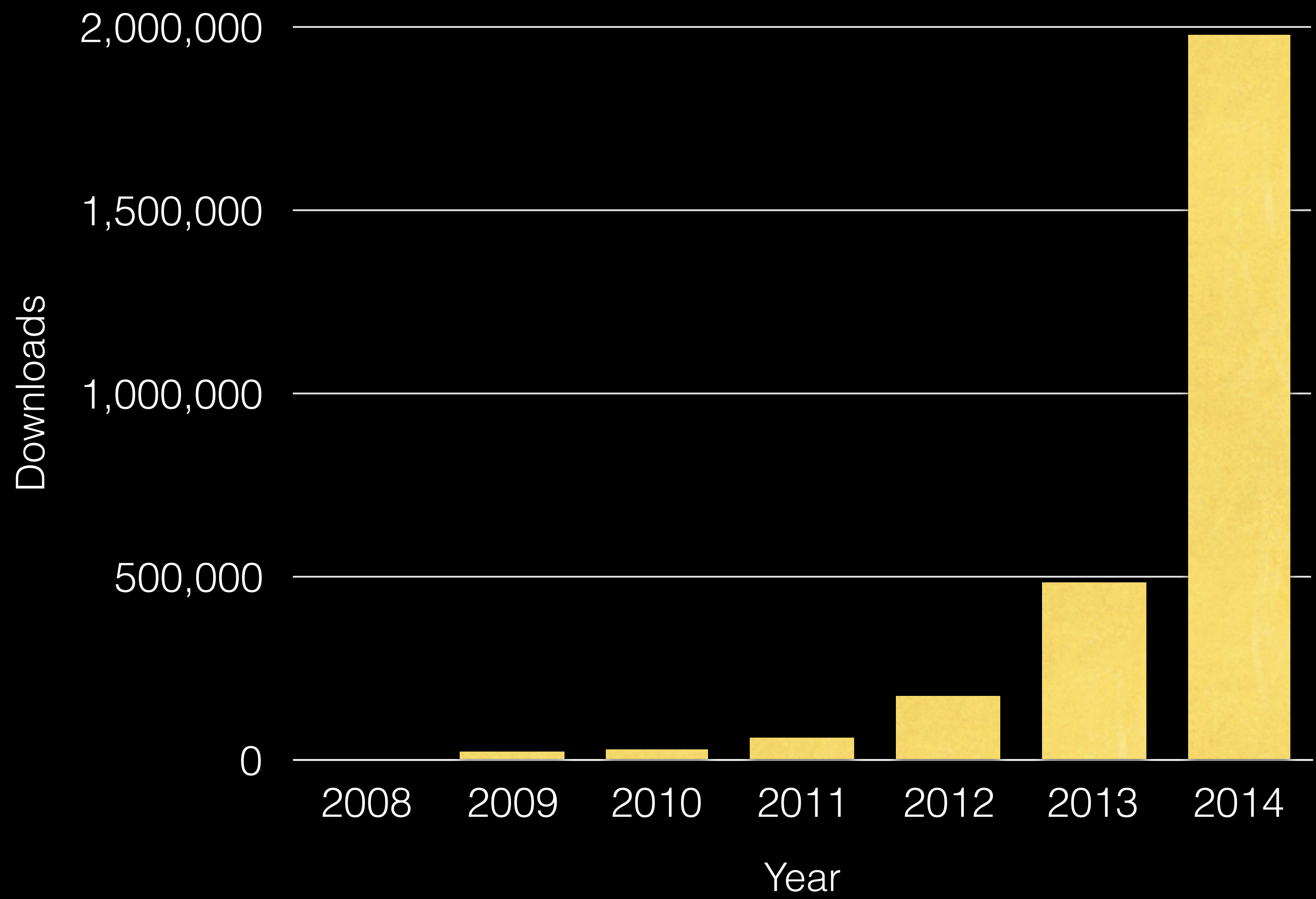
- **All chapters free online:** www.ostep.org

Material developed over 16 years @ Wisconsin

- First class notes in raw text files
- then added text figures
- then typeset
- then added better pictures
- then added homeworks
- then made printed copy available…

Hardcover: Print on Demand for ~$35

OSTEP: Chapter Downloads



FOB Conclusions

Stop charging for books!

- Too expensive, just funds publishers, not authors

Our goal: Free Online textBooks **(FOBs)**

- Share your knowledge with largest group possible

Current effort: **Free operating systems book**

- Operating Systems: Three Easy Pieces [www.ostep.org]
- Millions of chapter downloads … and hopes of writing a few more books in this style, as well as convincing others to do so!