Optimistic Crash Consistency

Vijay Chidambaram

CS 736 Graduate Operating Systems

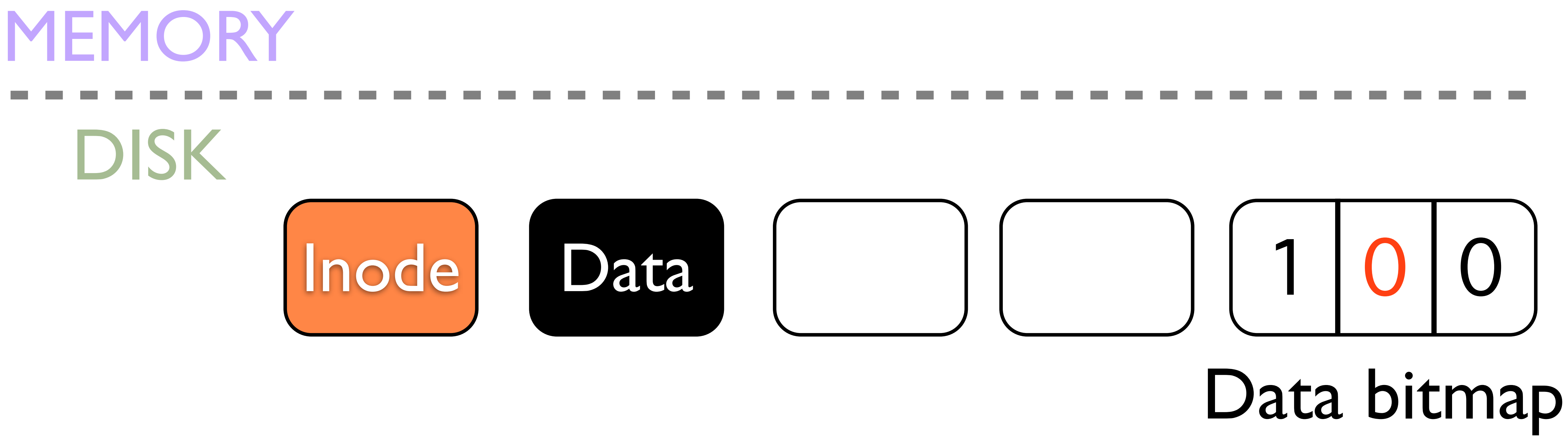
The Crash Consistency Problem

A single file-system operation updates multiple on-disk data structures

The system may crash in the middle of updating these structures

This leaves the file-system partially (incorrectly) updated

An Example



MEMORY

DISK

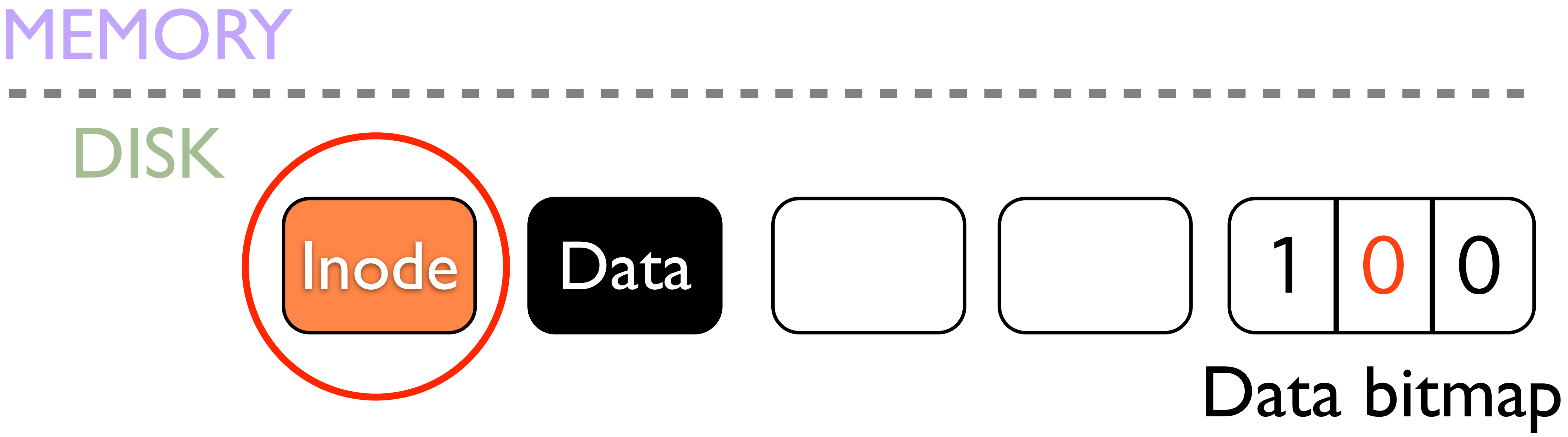
Inode

Data

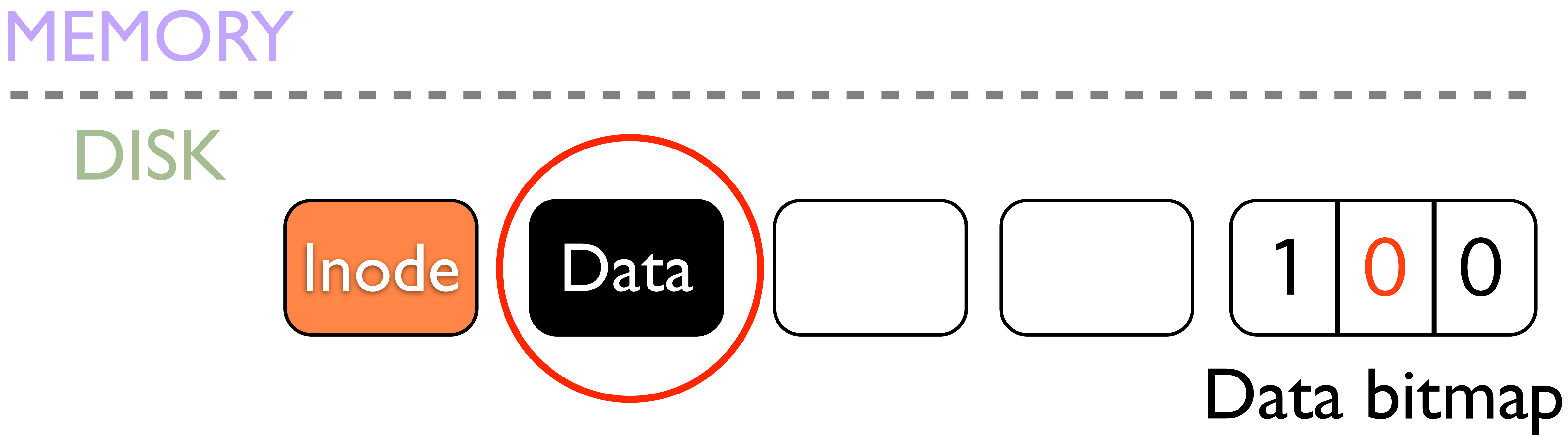
1 0 0

Data bitmap

An Example



An Example



MEMORY

DISK

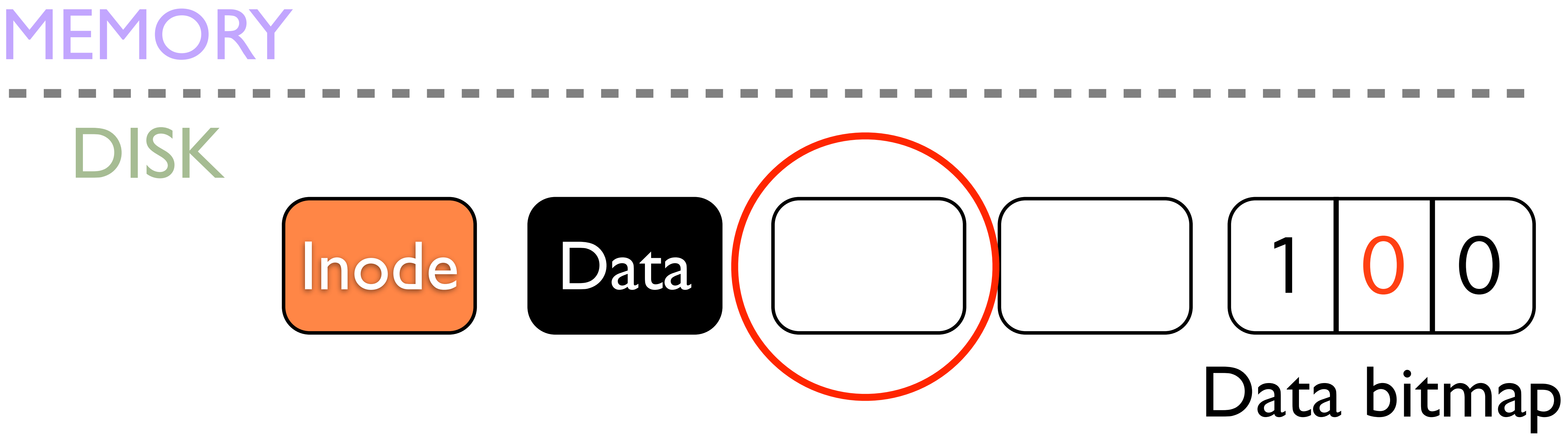
Inode

Data

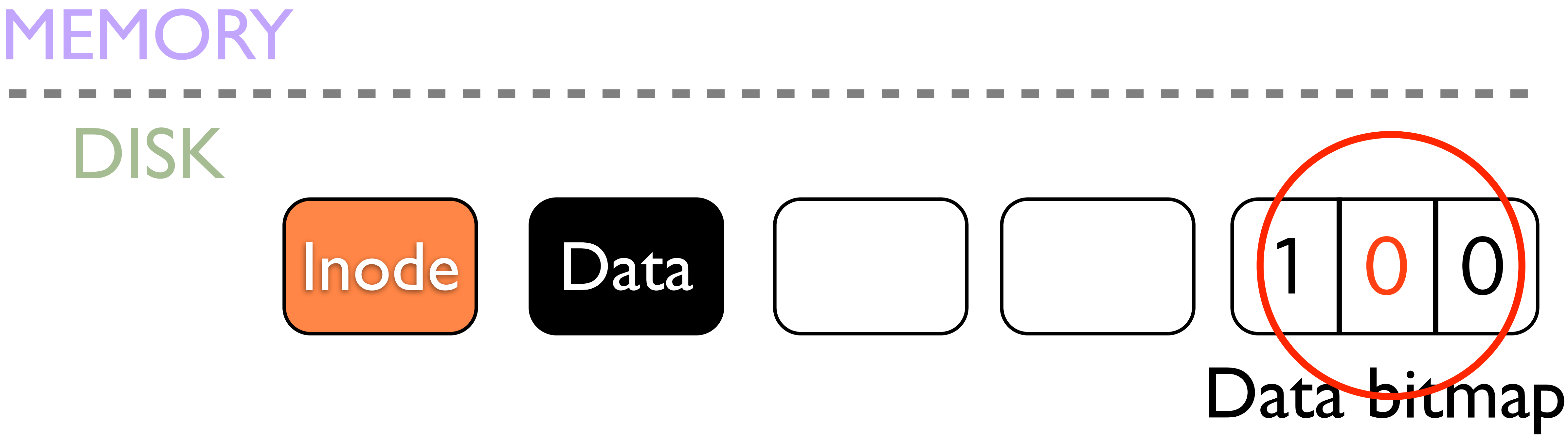
1 | 0 | 0

Data bitmap

An Example



An Example



MEMORY

DISK

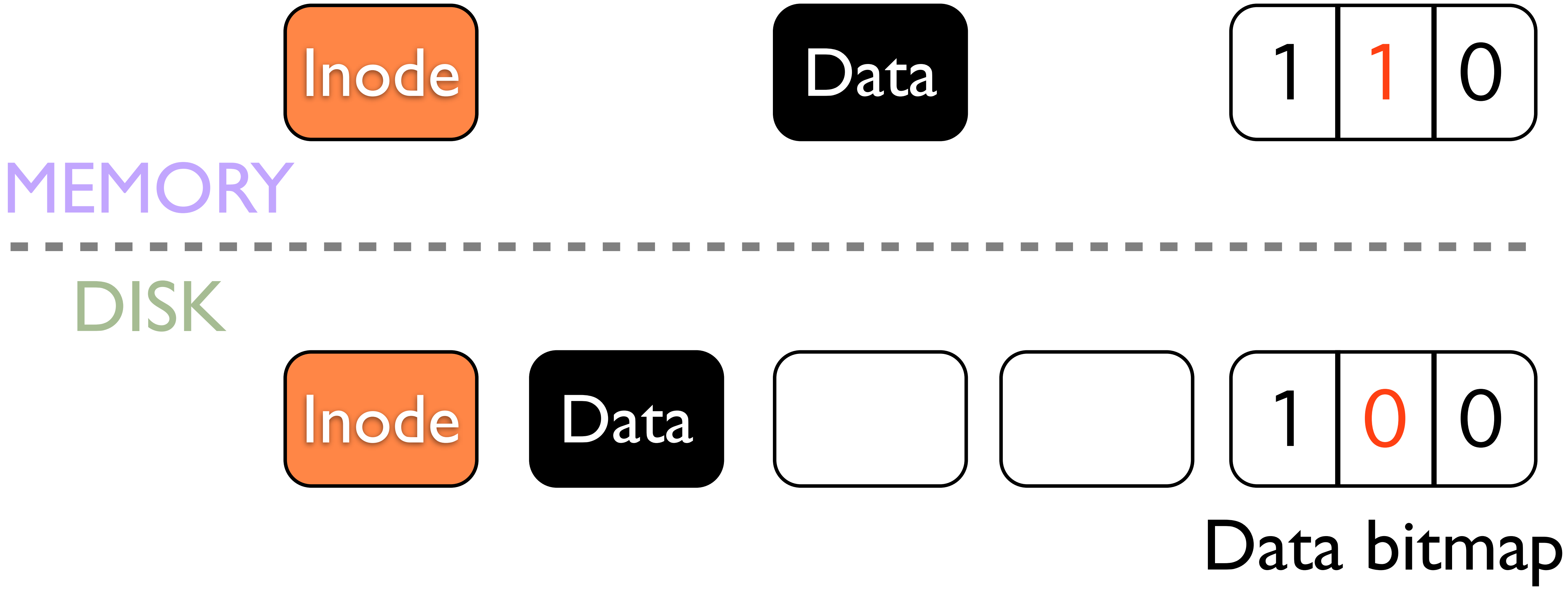
Inode

Data

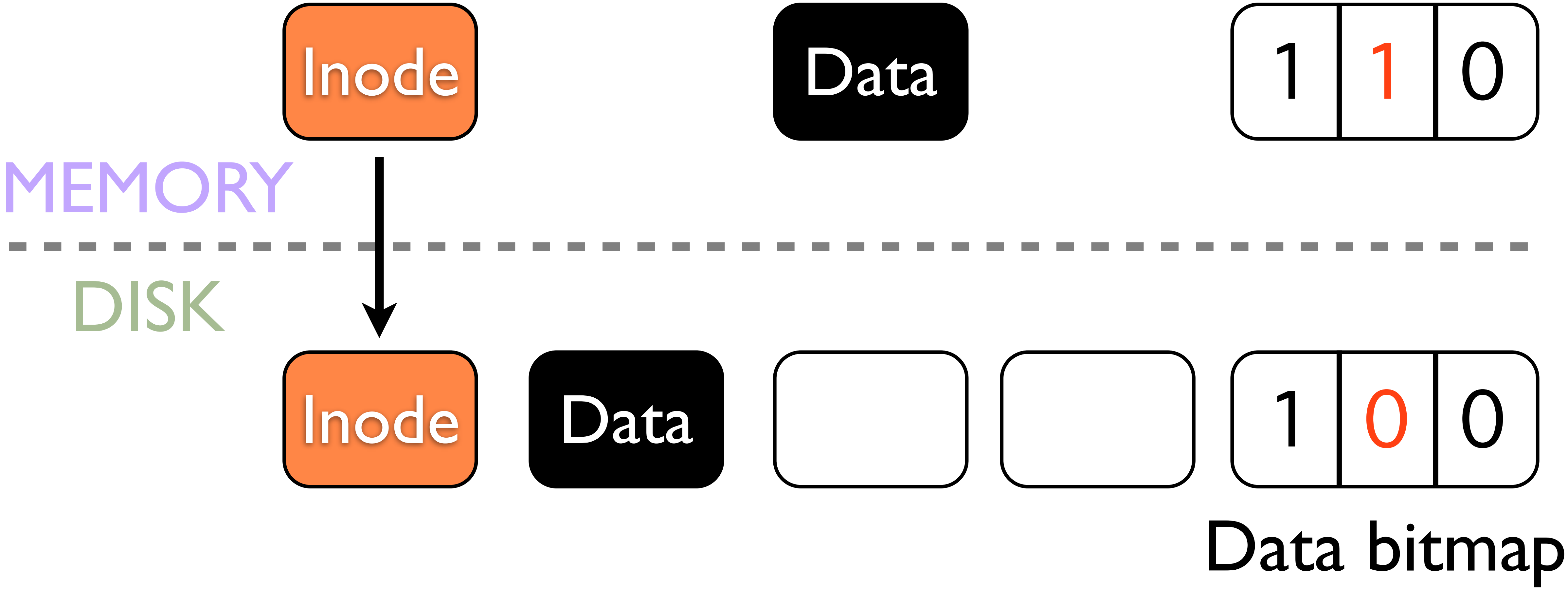
1 | 0 | 0

Data bitmap

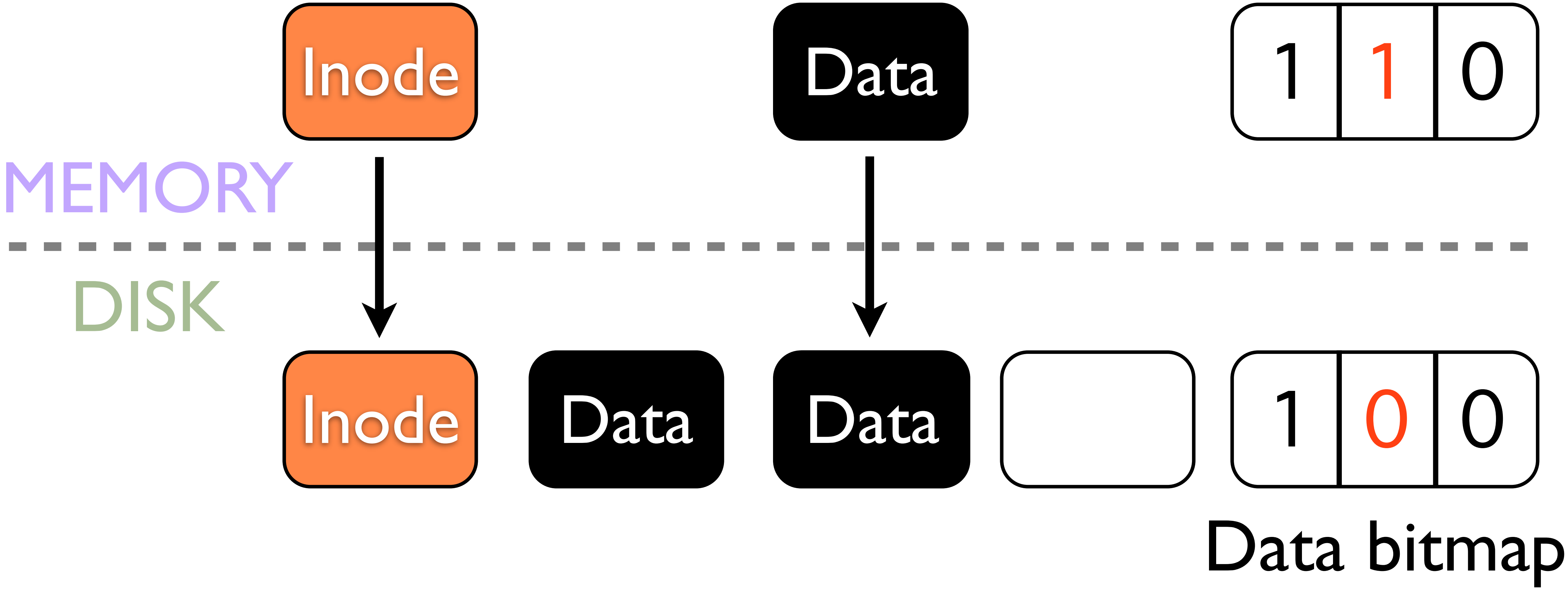
An Example



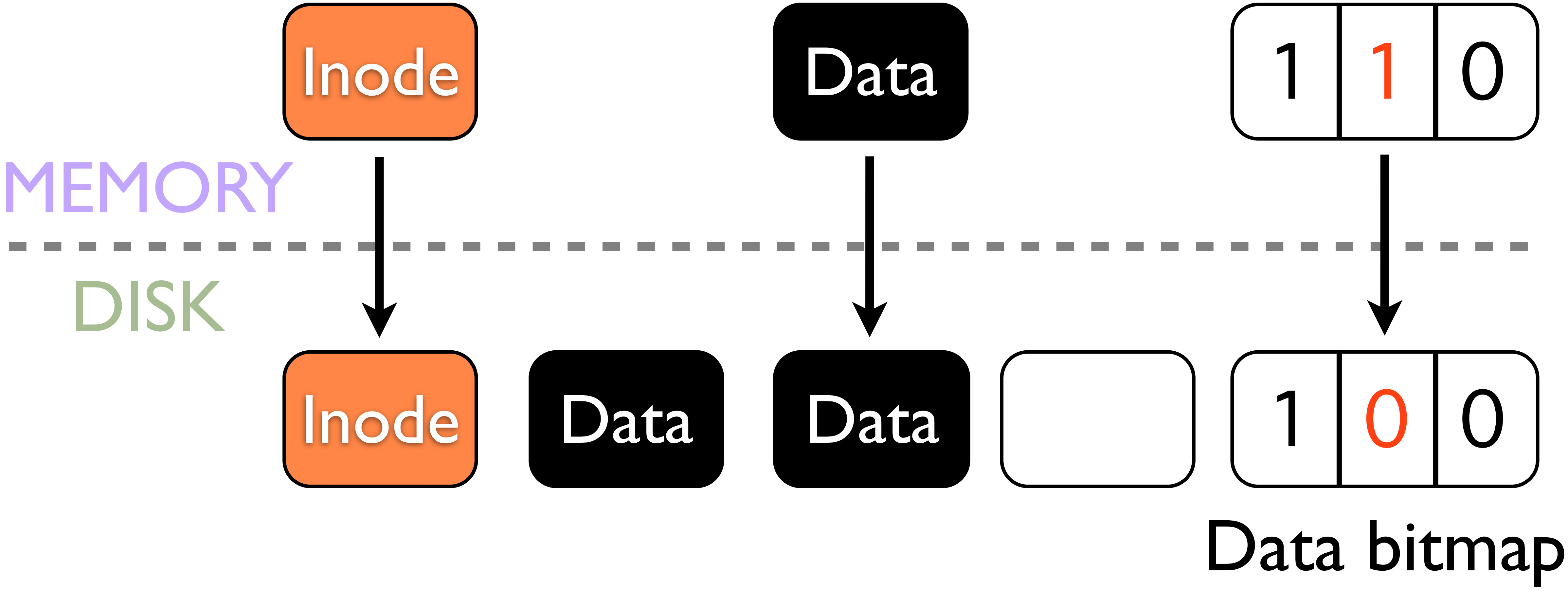
An Example



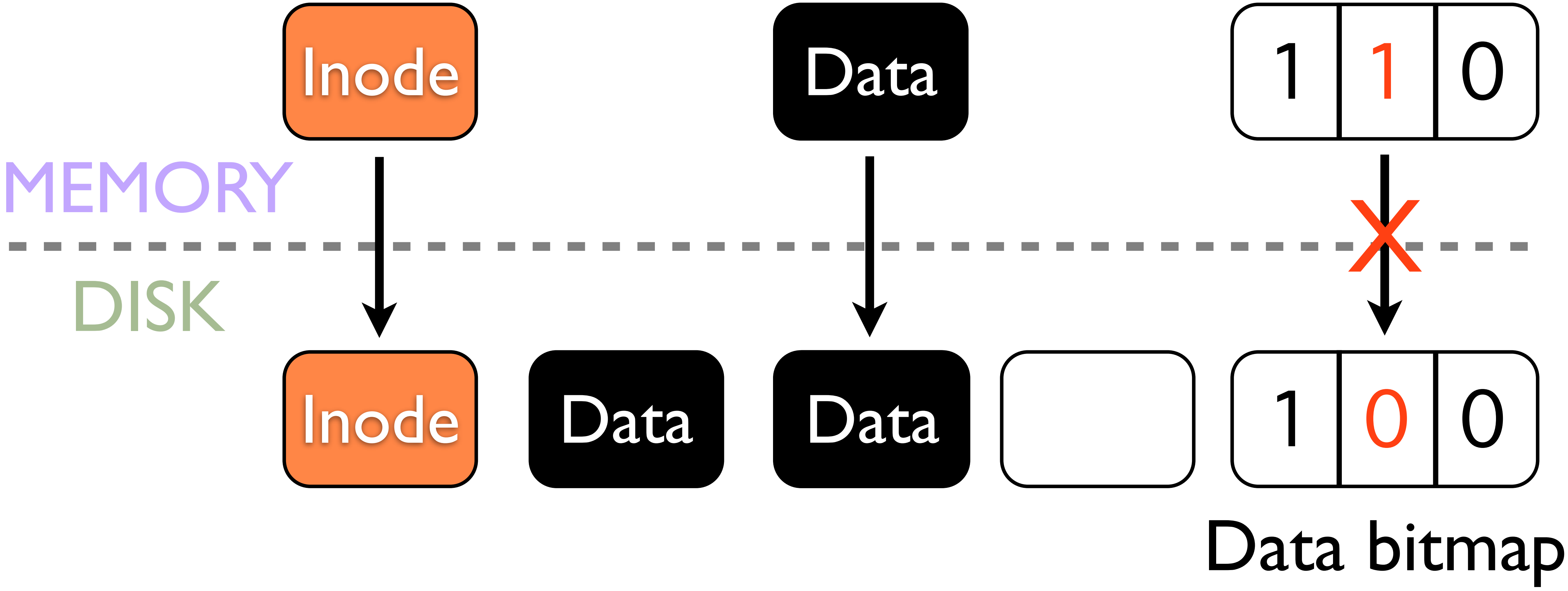
An Example



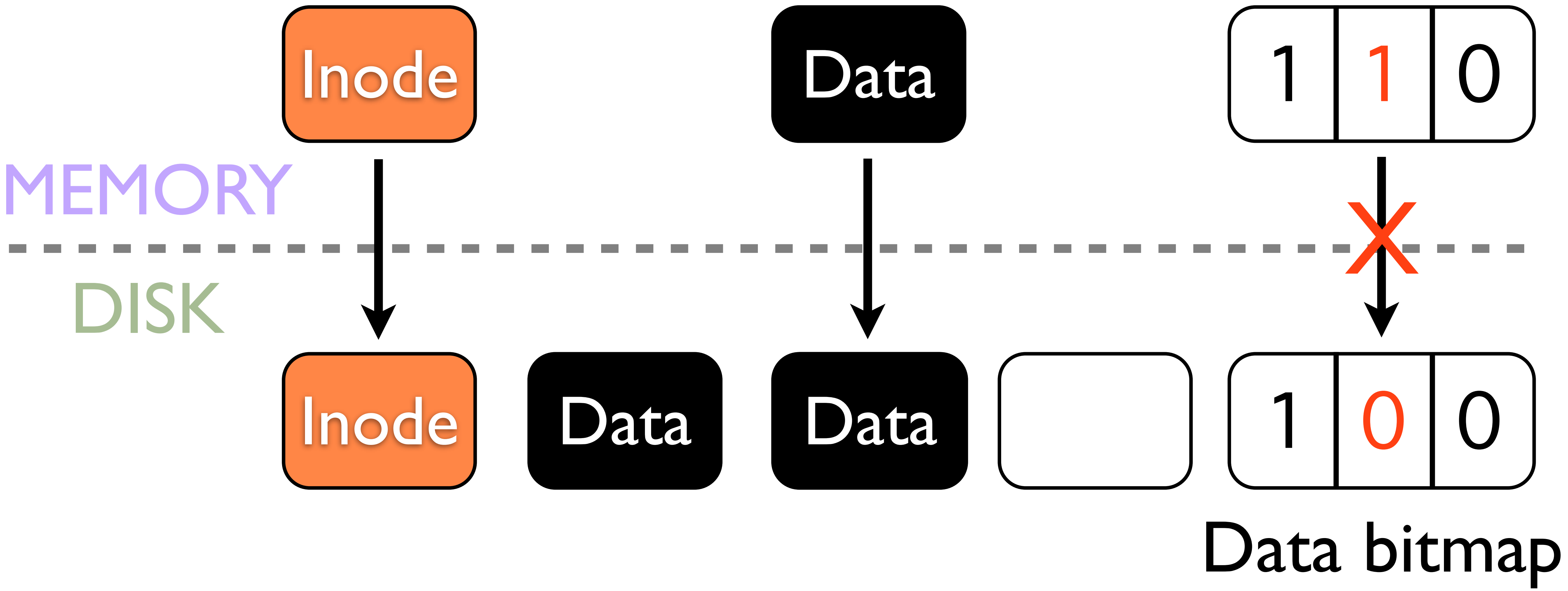
An Example



An Example



An Example



Problem: upon a crash, data structures on disk are partially updated

Current Solutions to Crash Consistency

Current Solutions to Crash Consistency

File-system check [McKusick84]

Journaling [Hagmann87]

Log structured file system [Rosenblum92]

Copy-on-write file system [Hitz94]

Soft Updates [Ganger94]

Journaling

Journaling

Before updating the file system, write a note describing the update first

Journaling

Before updating the file system, write a note describing the update first

Make sure note is safely on disk

Journaling

Before updating the file system, write a note describing the update first

Make sure note is safely on disk

Once the note is safe, update the file system

Journaling

Before updating the file system, write a note describing the update first

Make sure note is safely on disk

Once the note is safe, update the file system

If the above step is interrupted, read note and do step again

Journaling

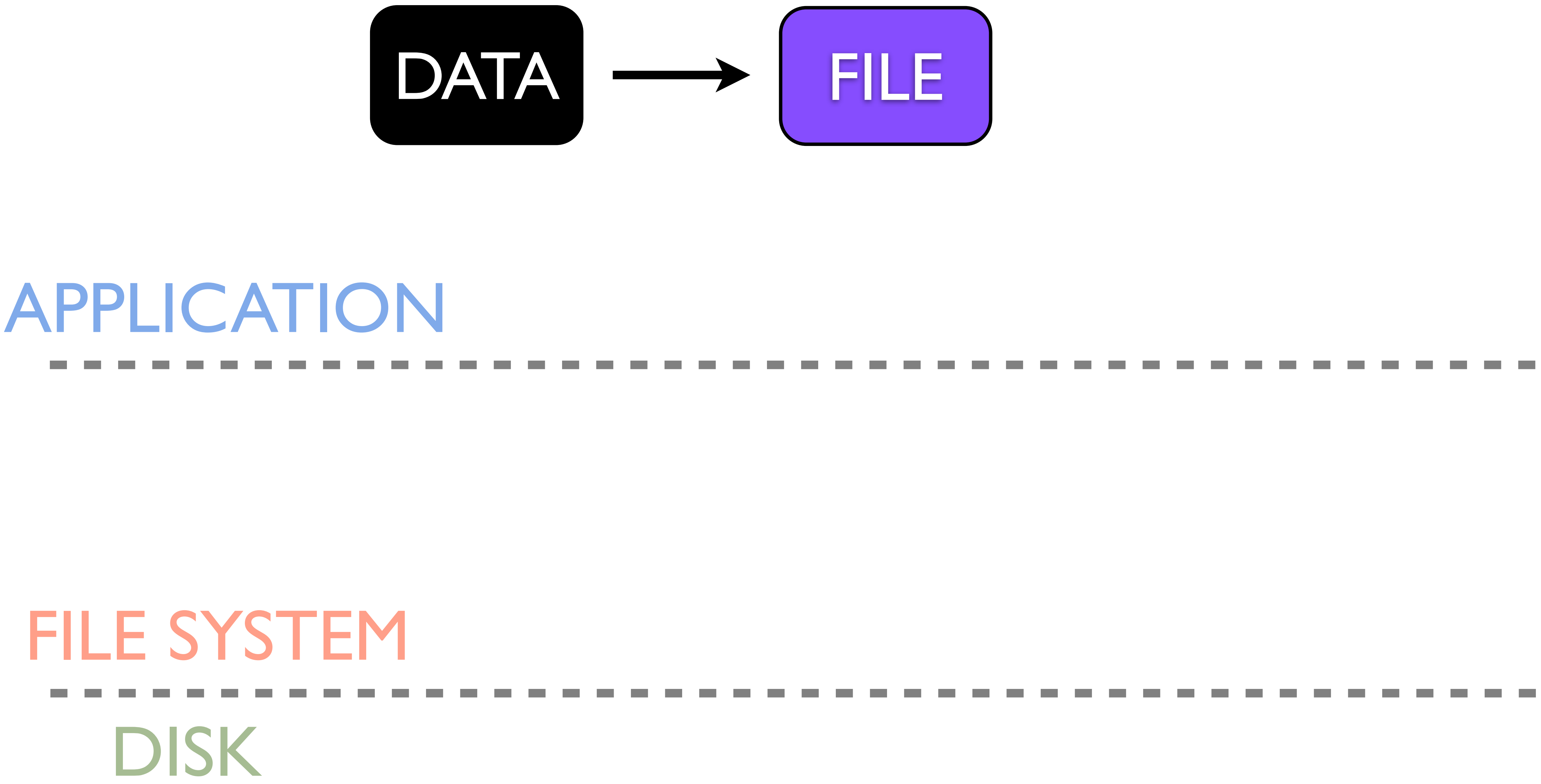
Before updating the file system, write a note describing the update first

Make sure note is safely on disk

Once the note is safe, update the file system

If the above step is interrupted, read note and do step again

Journaling: an example



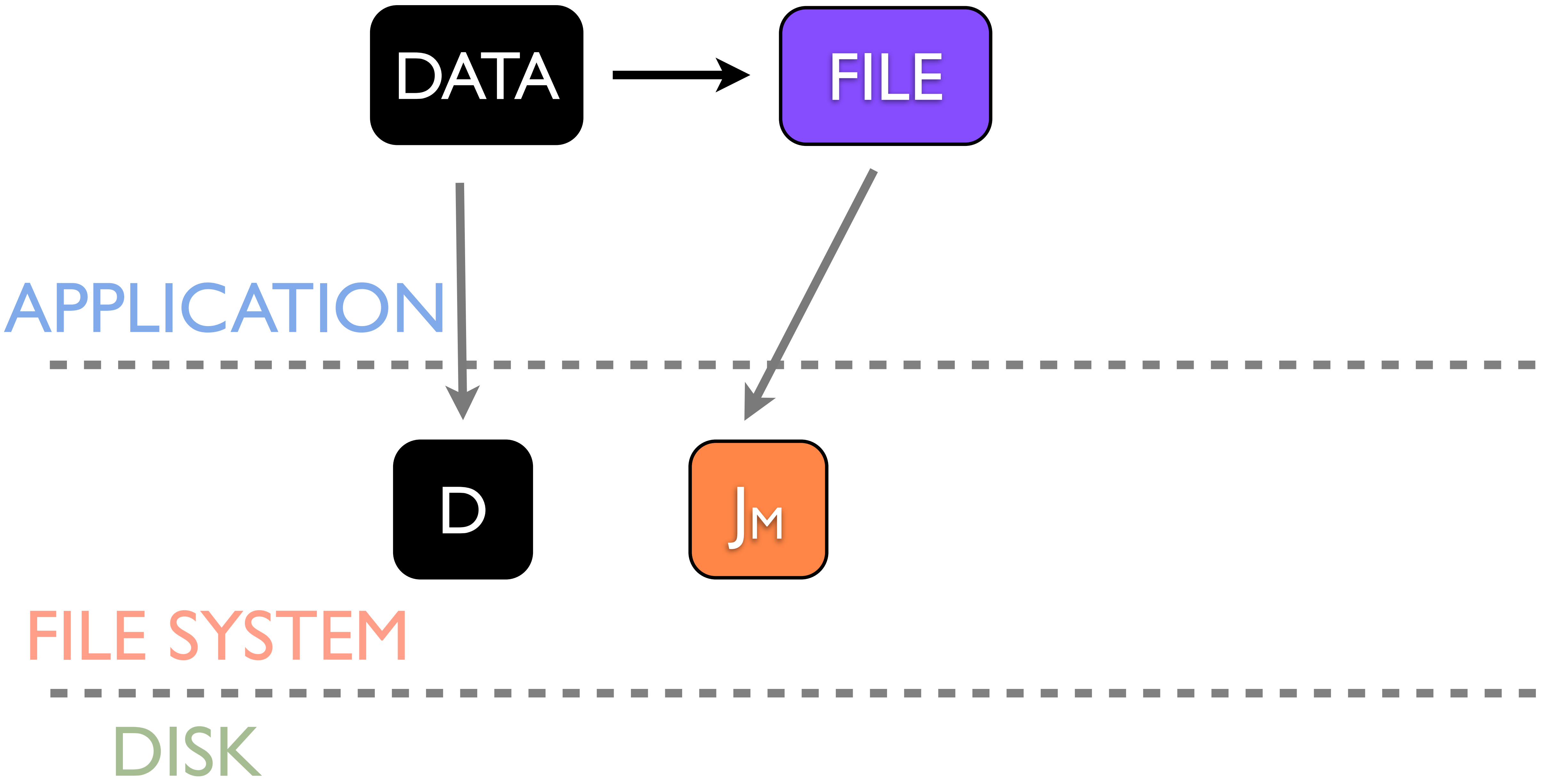
DATA → FILE

APPLICATION

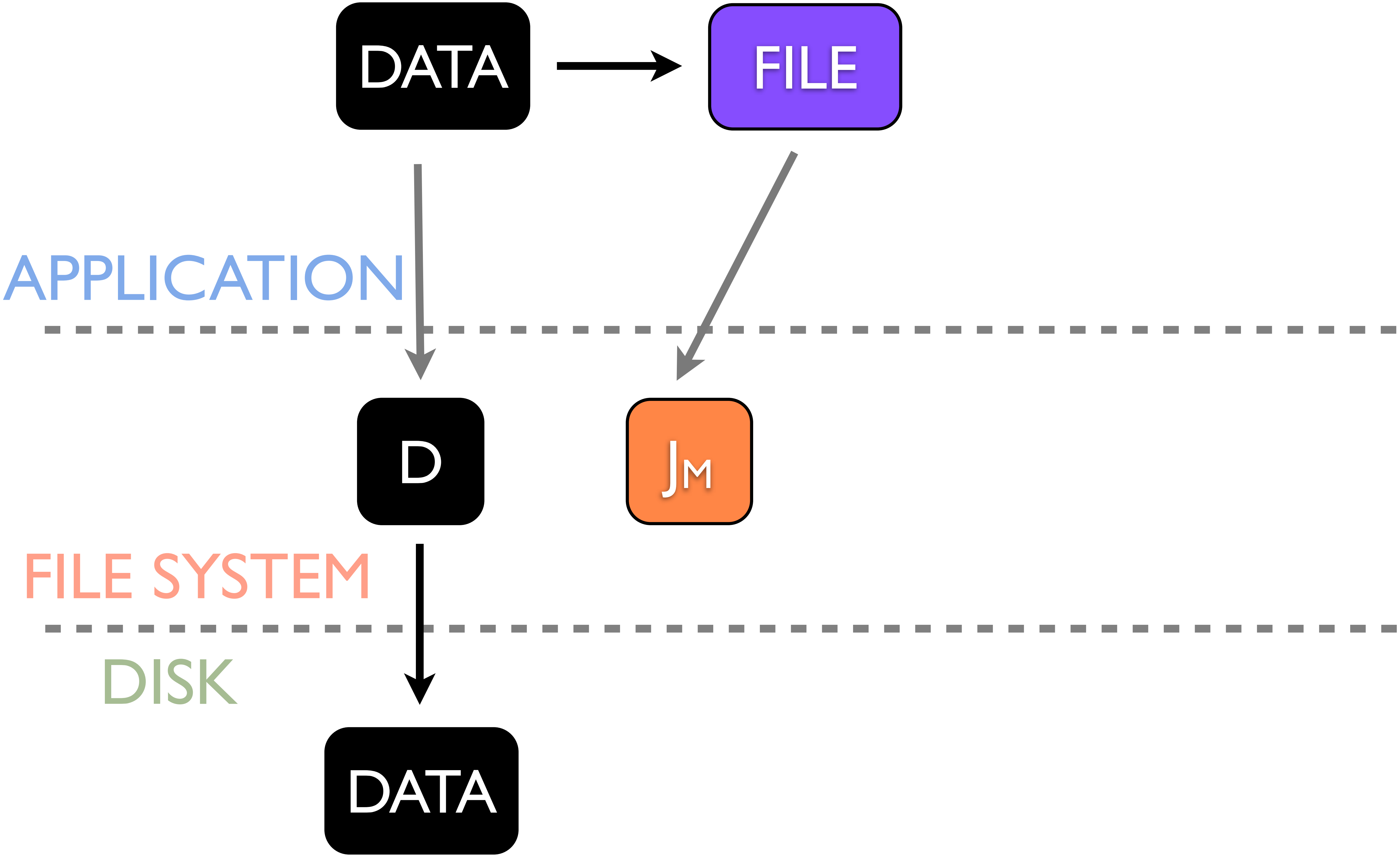
FILE SYSTEM

DISK

Journaling: an example



Journaling: an example



DATA

FILE

APPLICATION

D

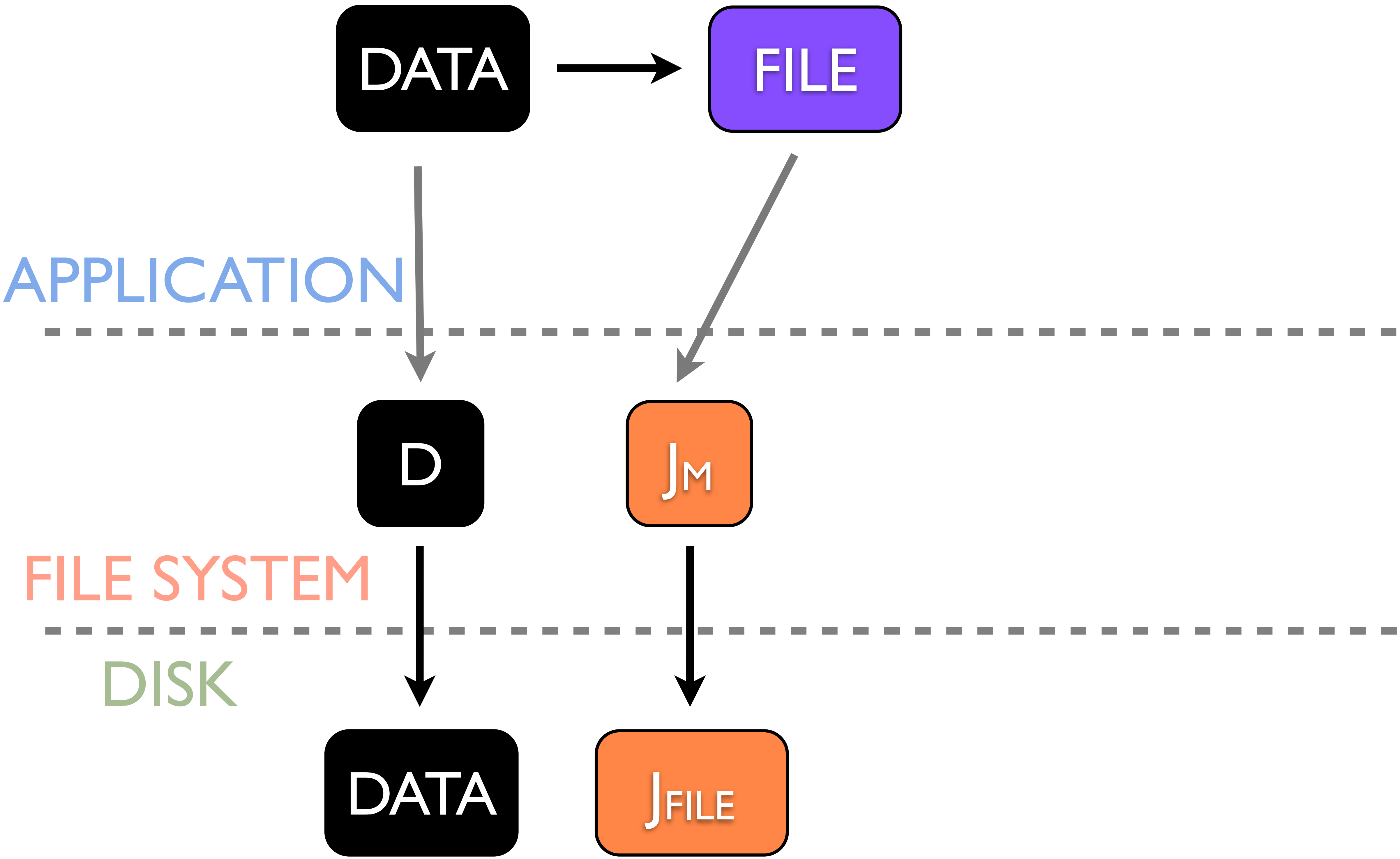
JM

FILE SYSTEM

DISK

DATA

Journaling: an example



DATA → FILE

APPLICATION

D

JM

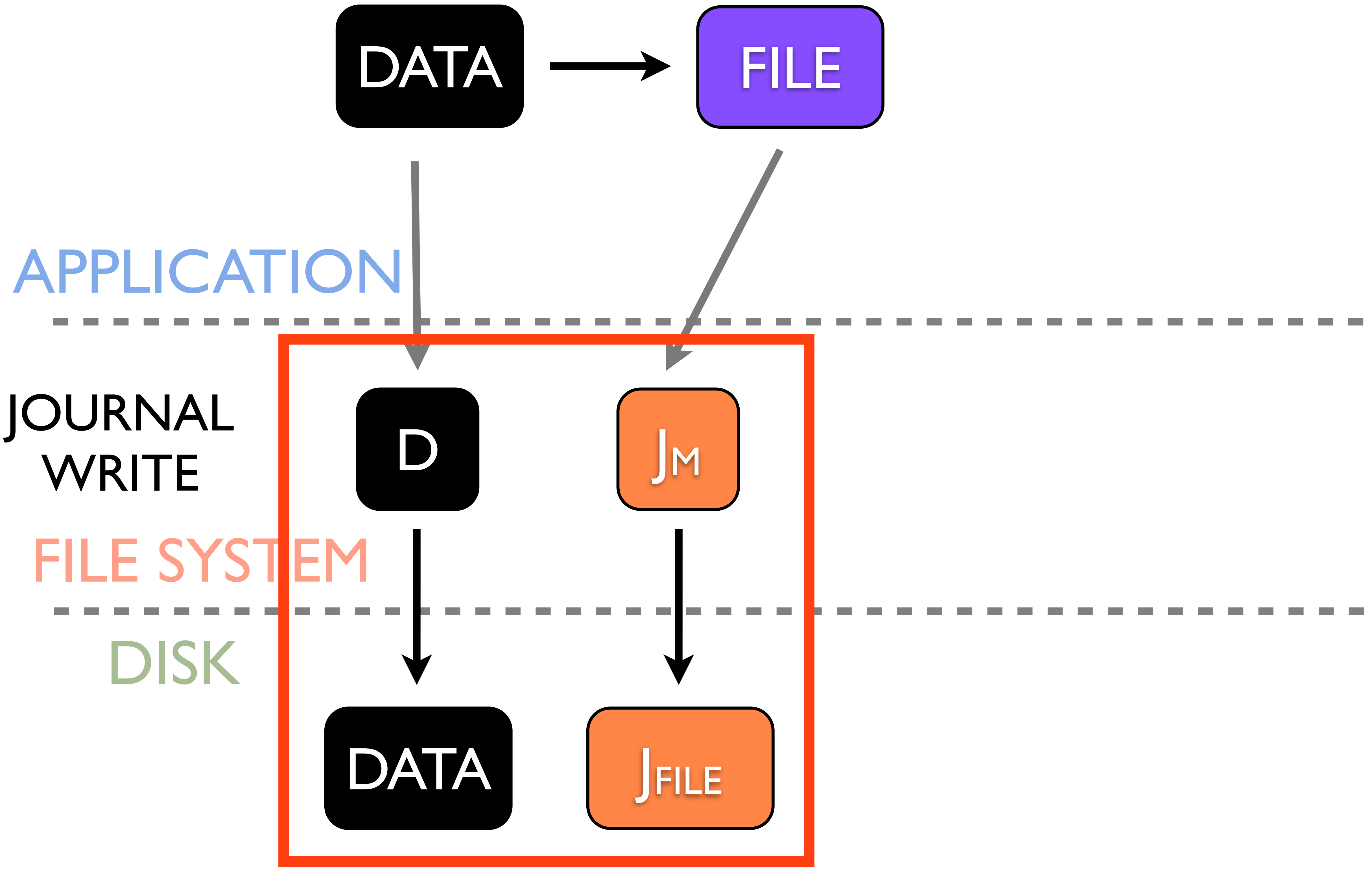
FILE SYSTEM

DISK

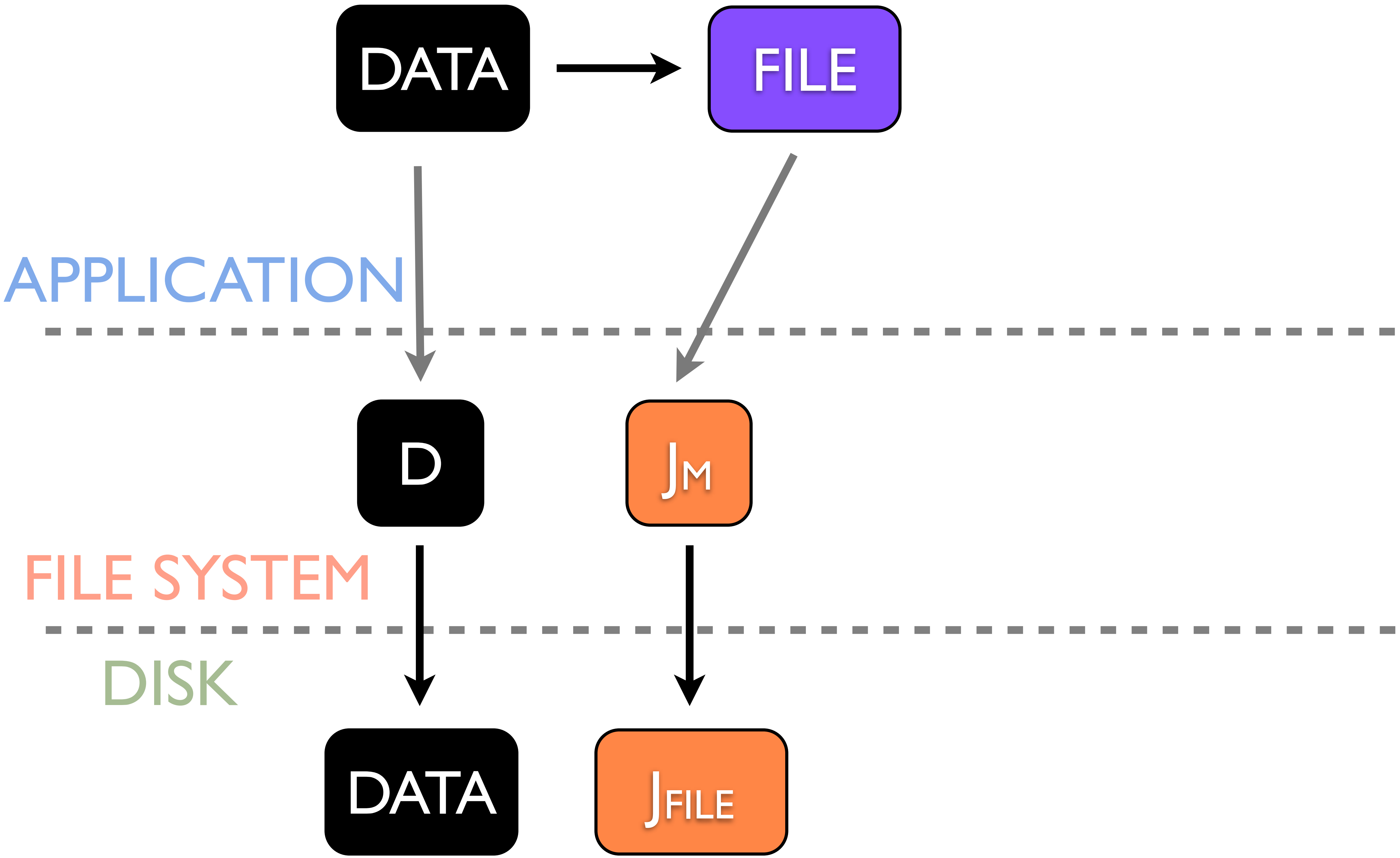
DATA

JFILE

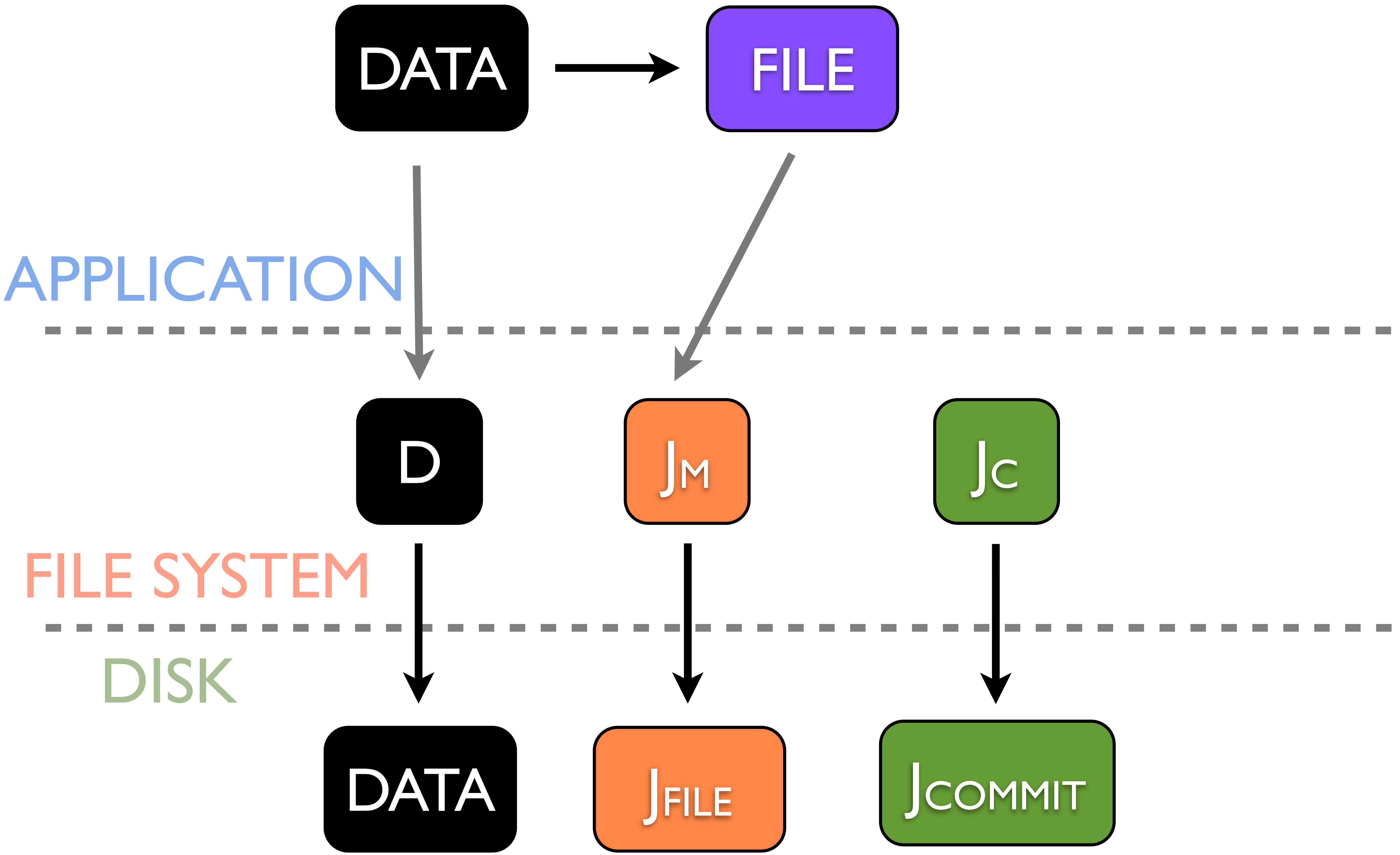
Journaling: an example



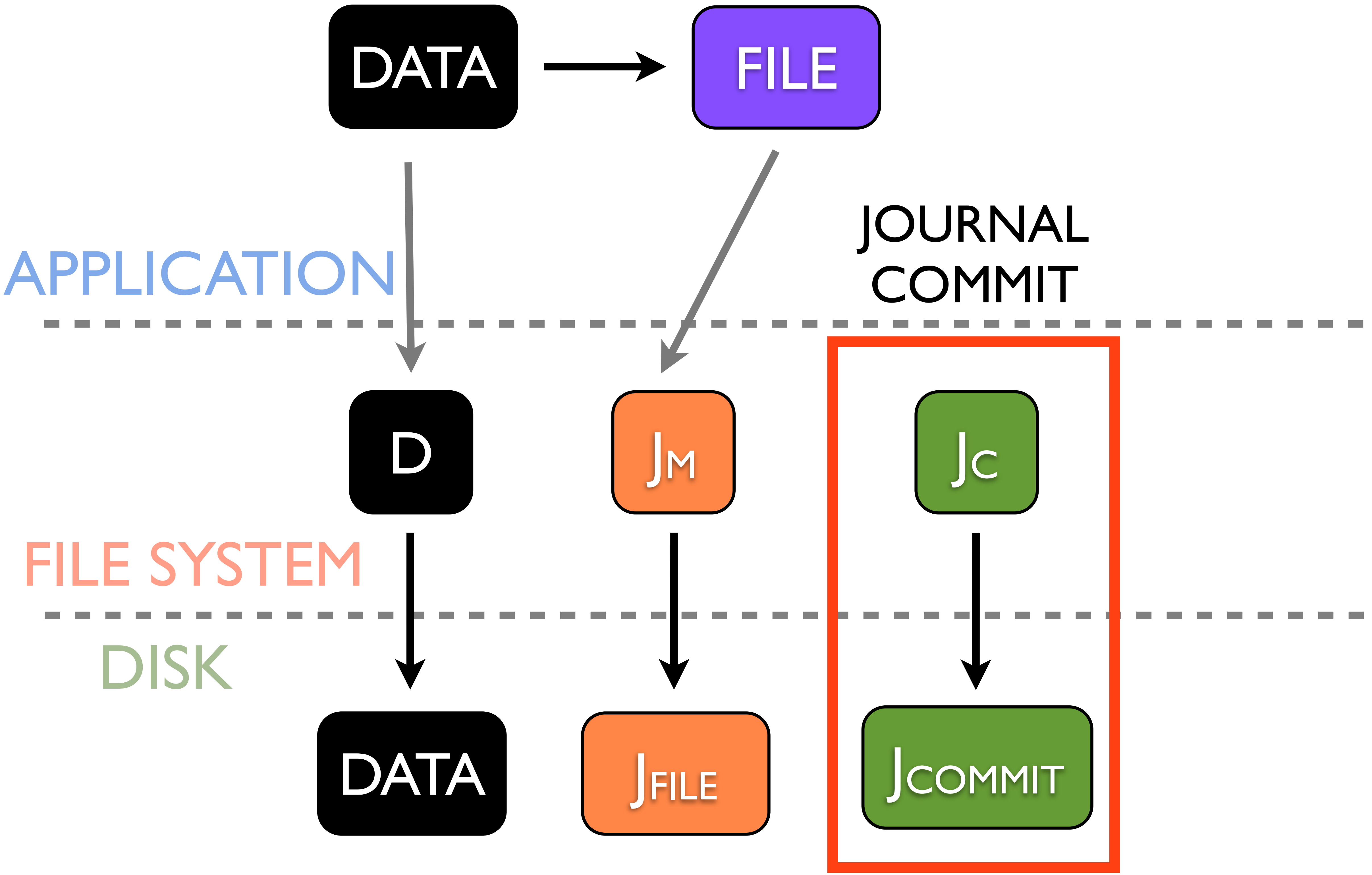
Journaling: an example



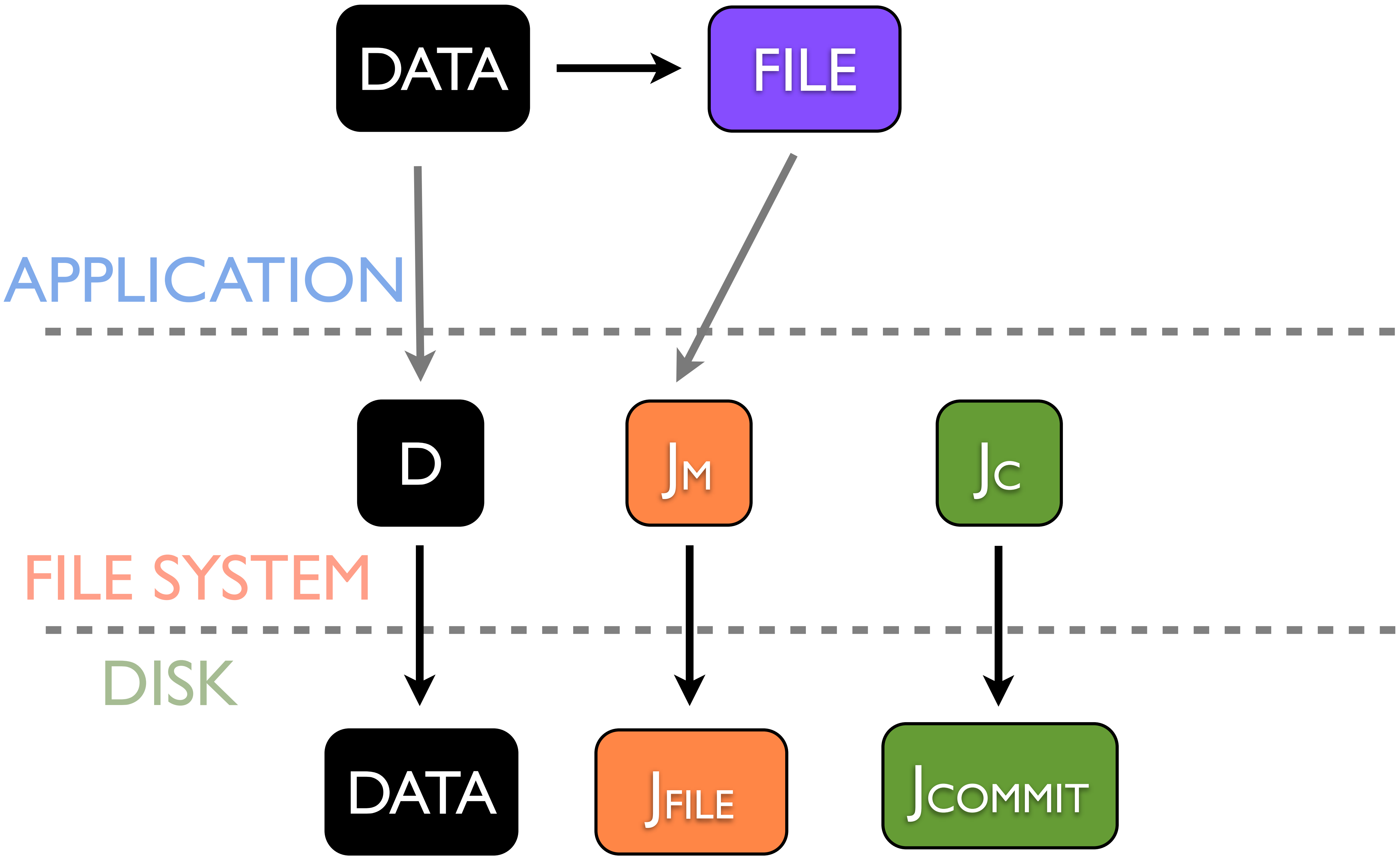
Journaling: an example



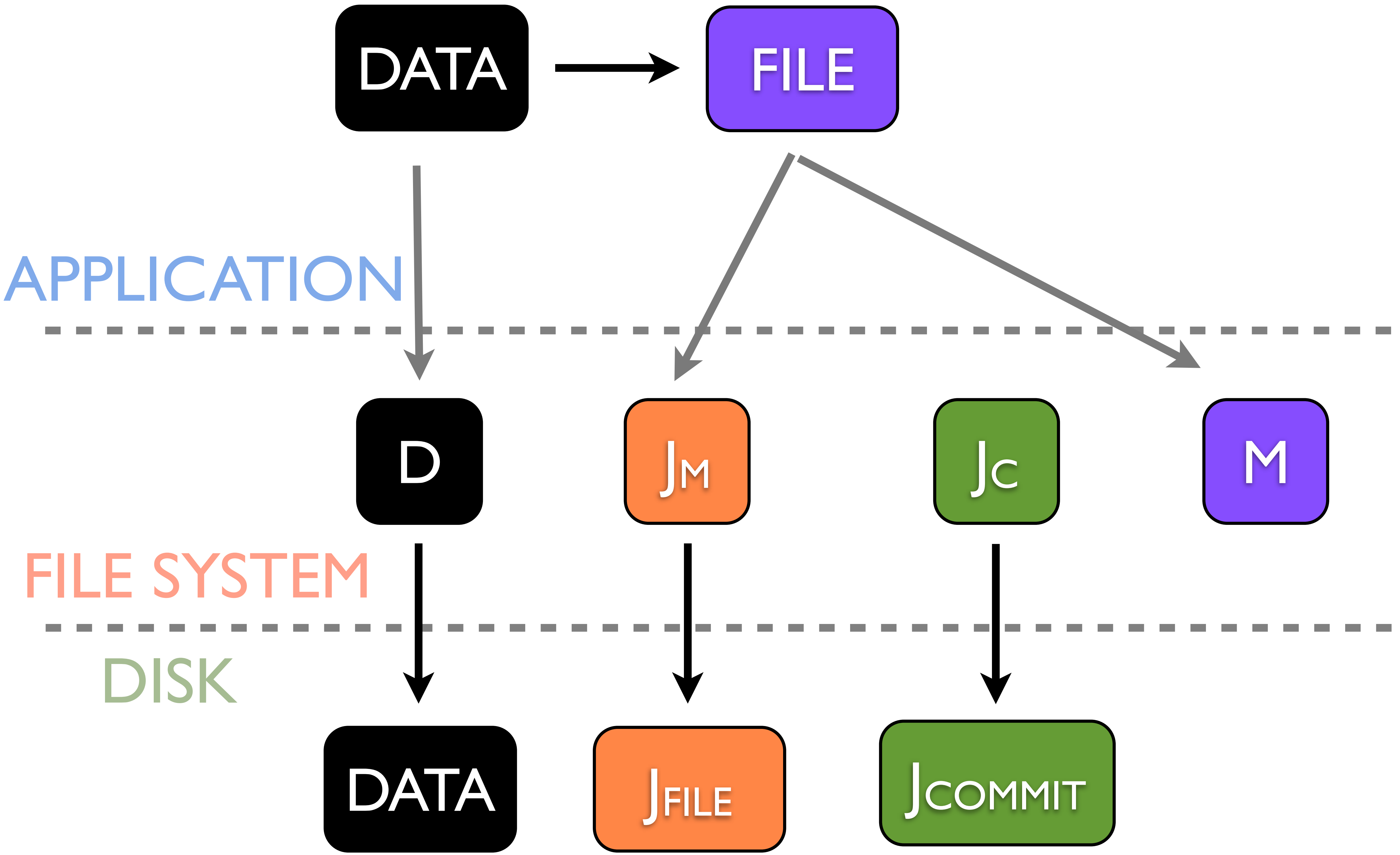
Journaling: an example



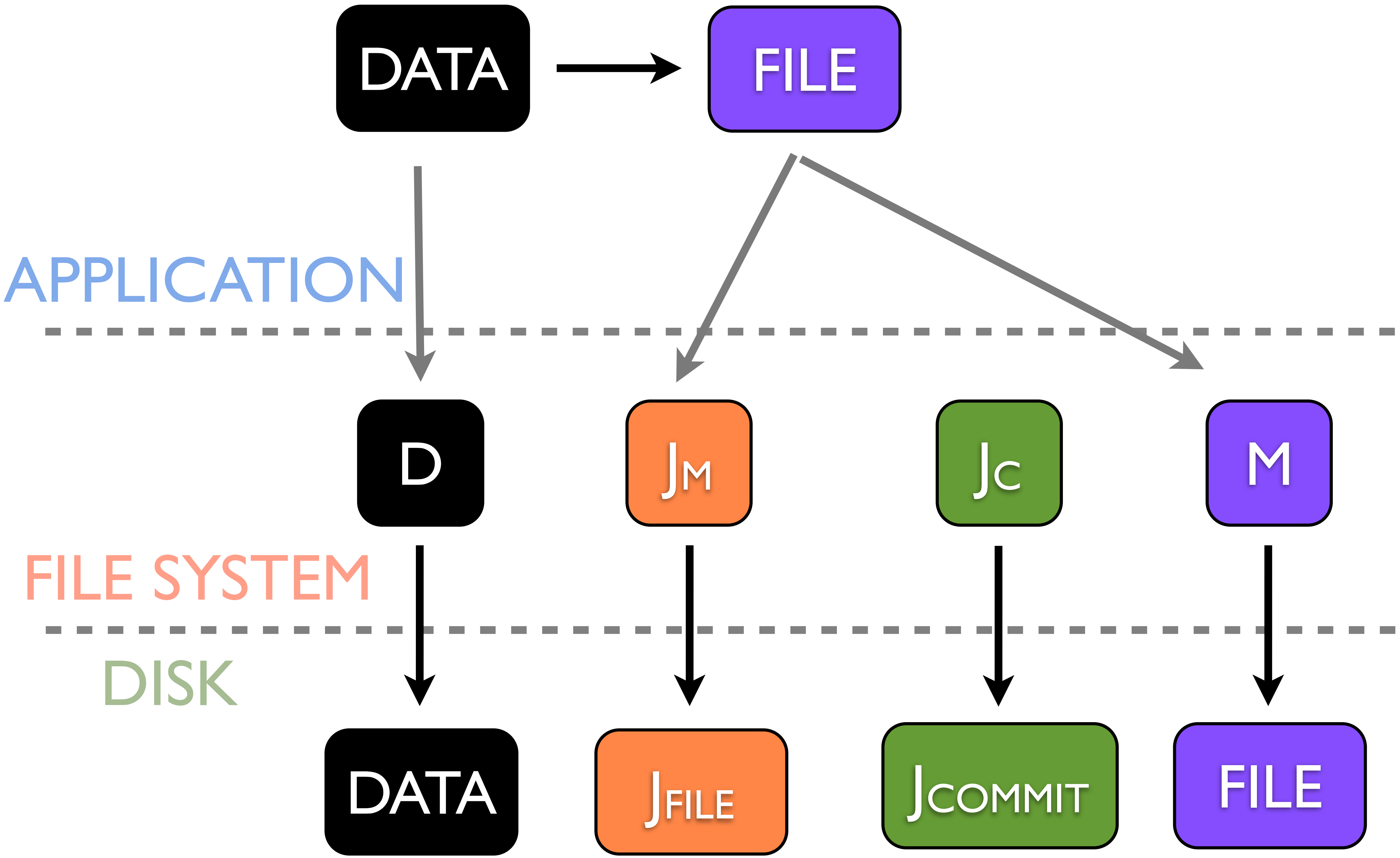
Journaling: an example



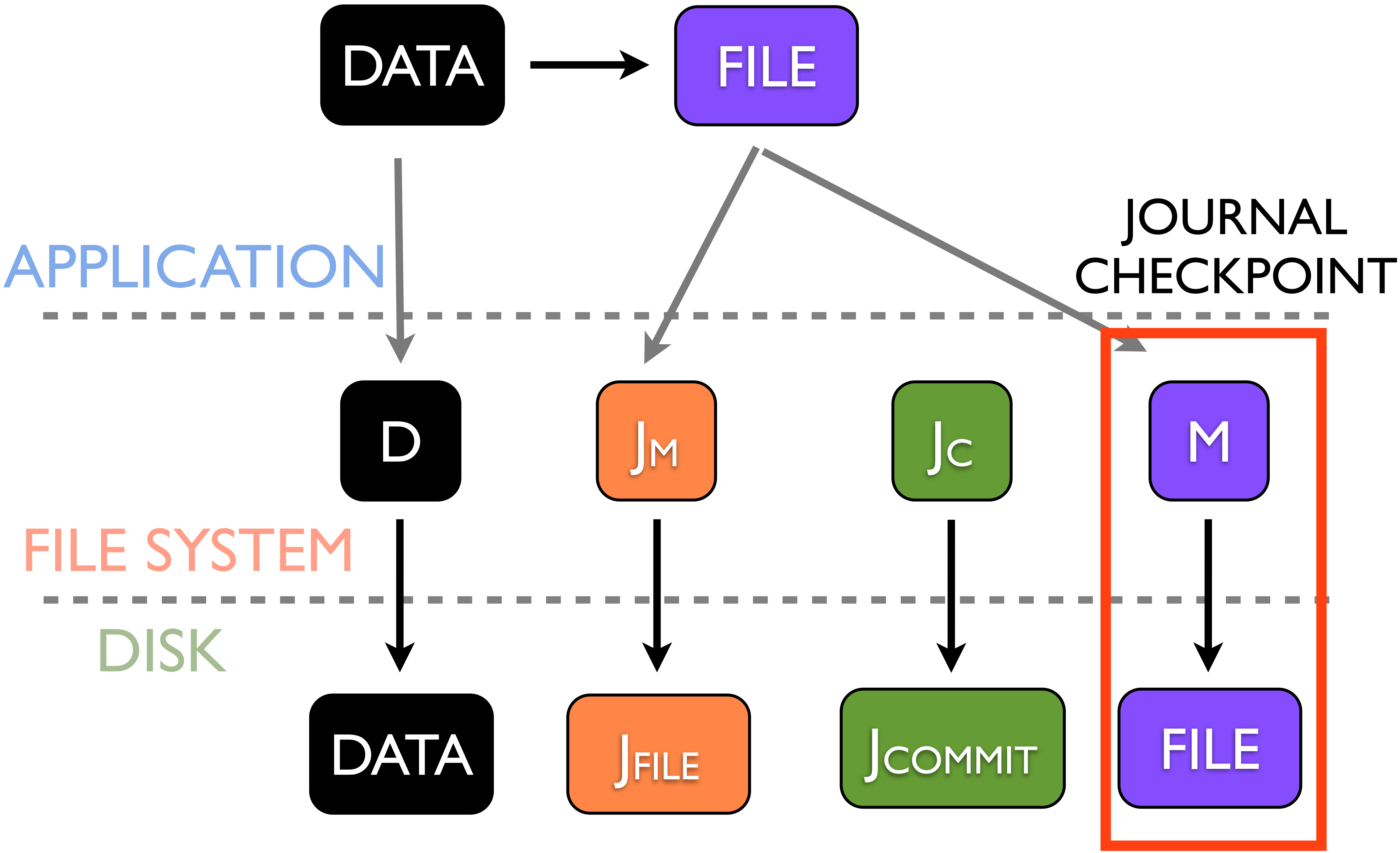
Journaling: an example



Journaling: an example



Journaling: an example



Ordered Writes

Journaling is built upon writing to disk in the correct order:

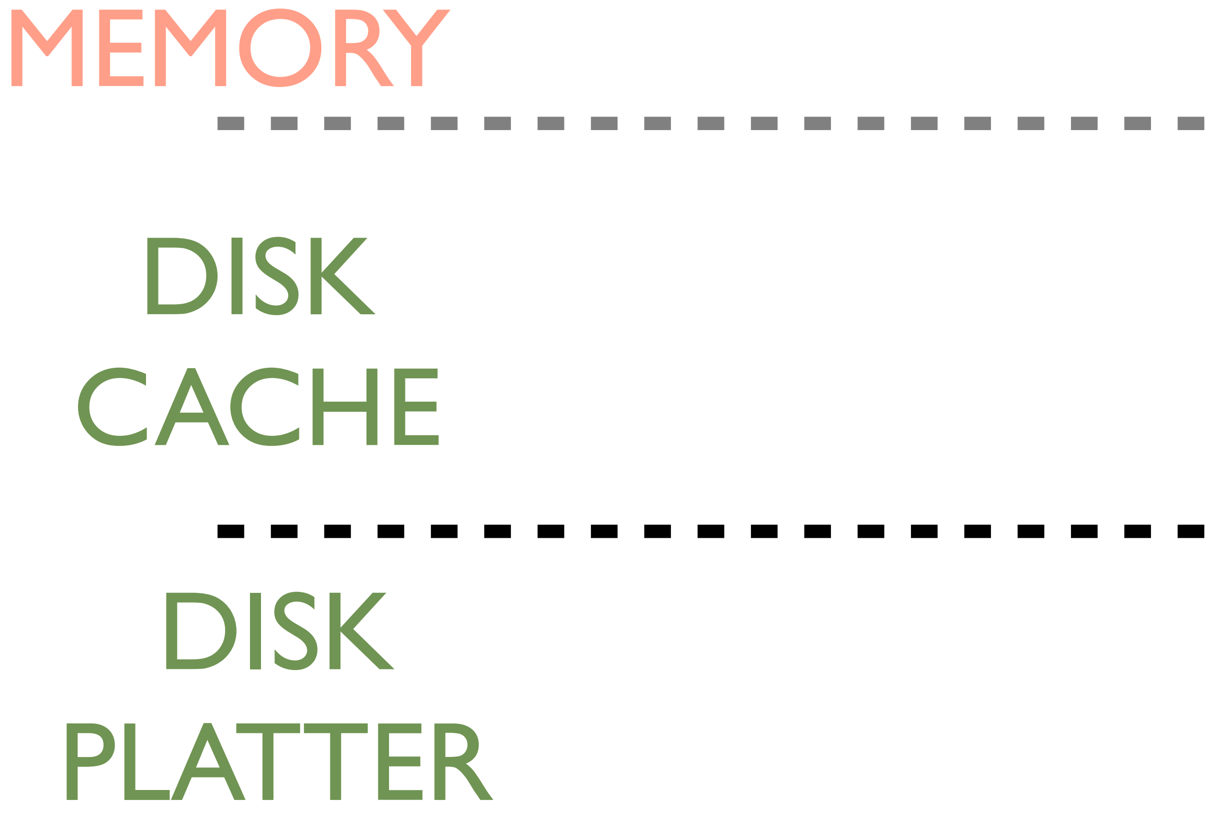
- Journal Writes
- Journal Commit
- Journal Checkpointing

Ex: if checkpointing happens before commit and transaction fails, file system is corrupted

Ordering Writes in Disks

Modern disk drives have on-board RAM caches

Writes are first buffered, then destaged to the non-volatile platter


MEMORY

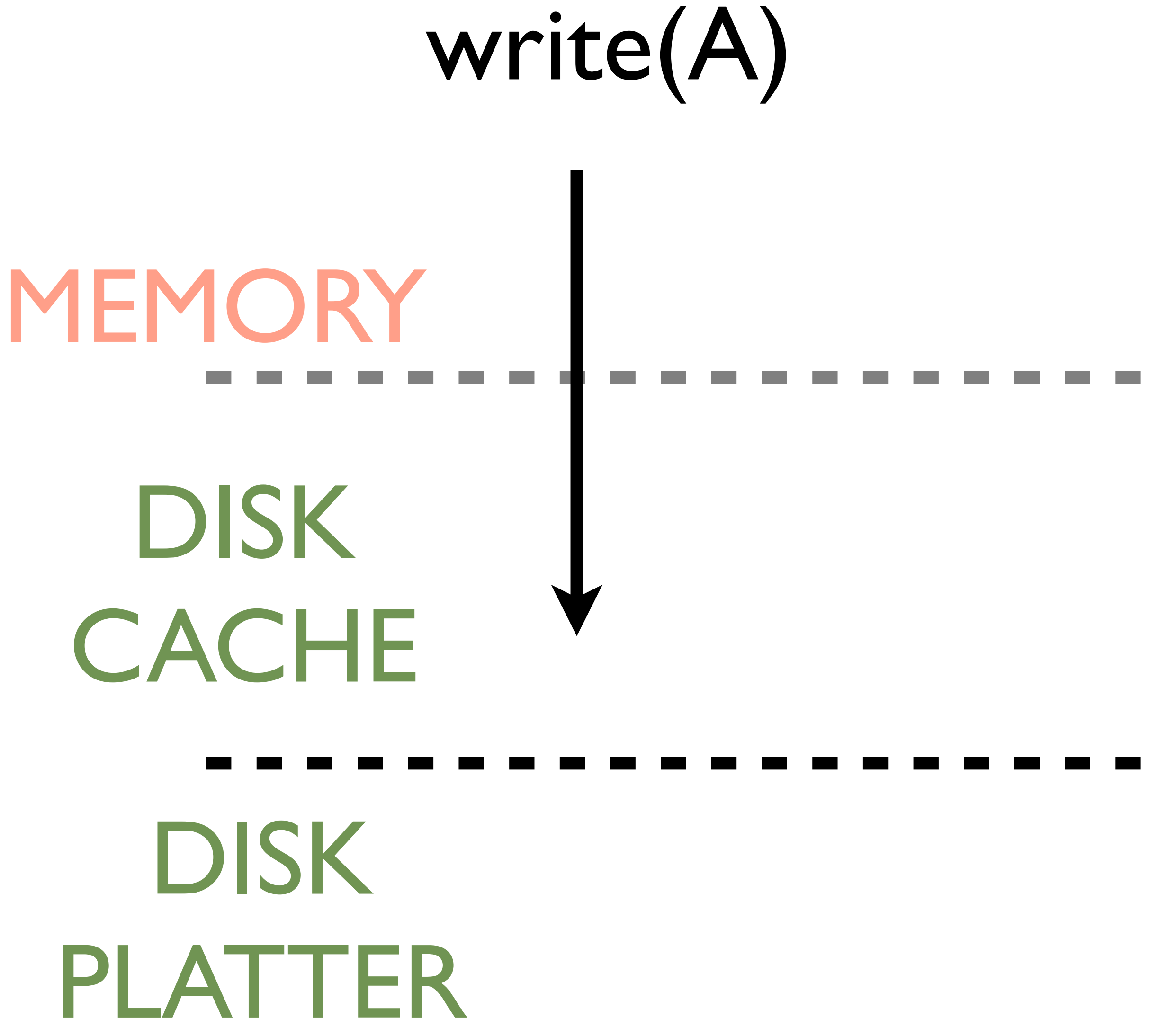
DISK CACHE

DISK PLATTER

Ordering Writes in Disks

Modern disk drives have on-board RAM caches

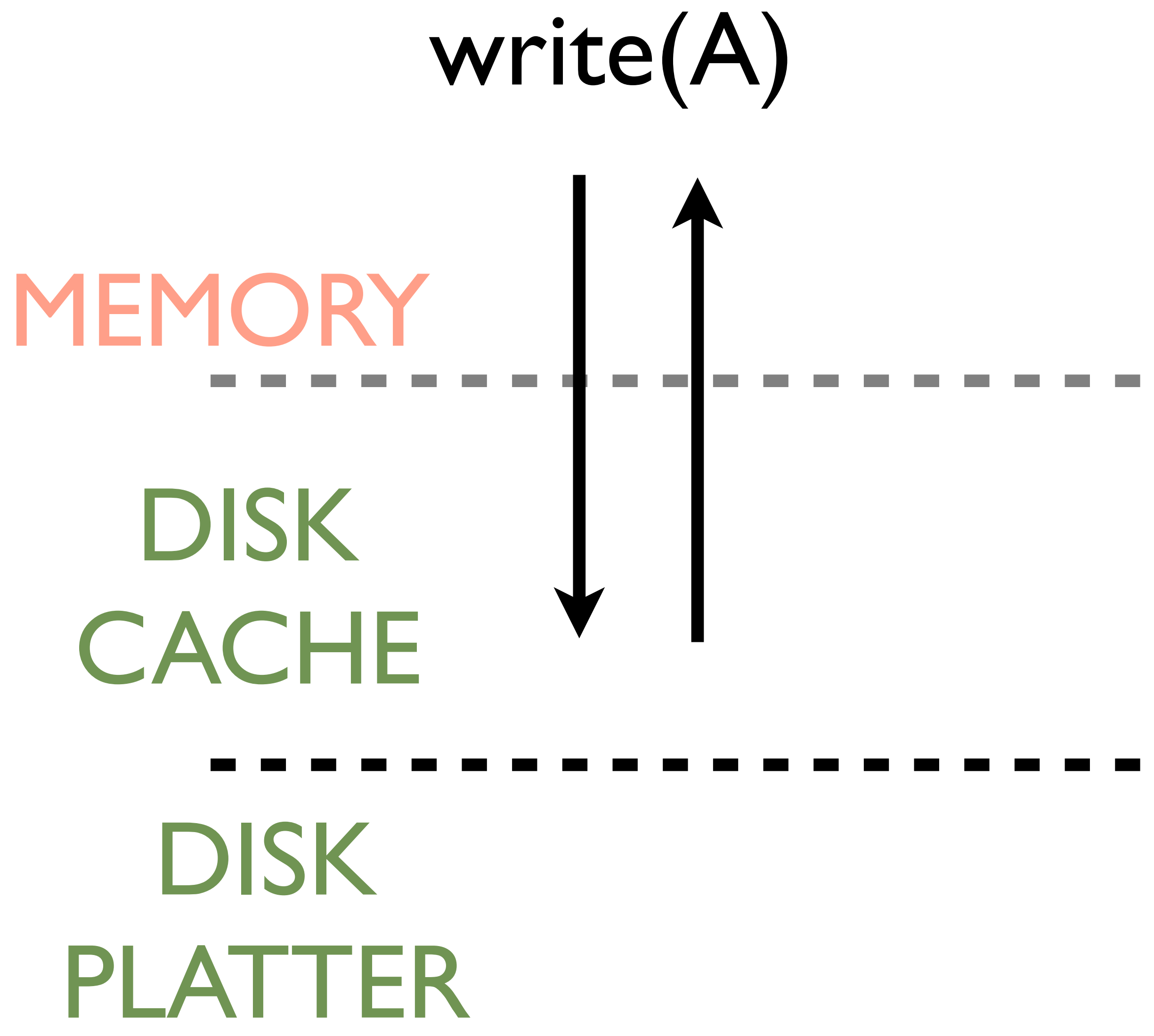
Writes are first buffered, then destaged to the non-volatile platter



Ordering Writes in Disks

Modern disk drives have on-board RAM caches

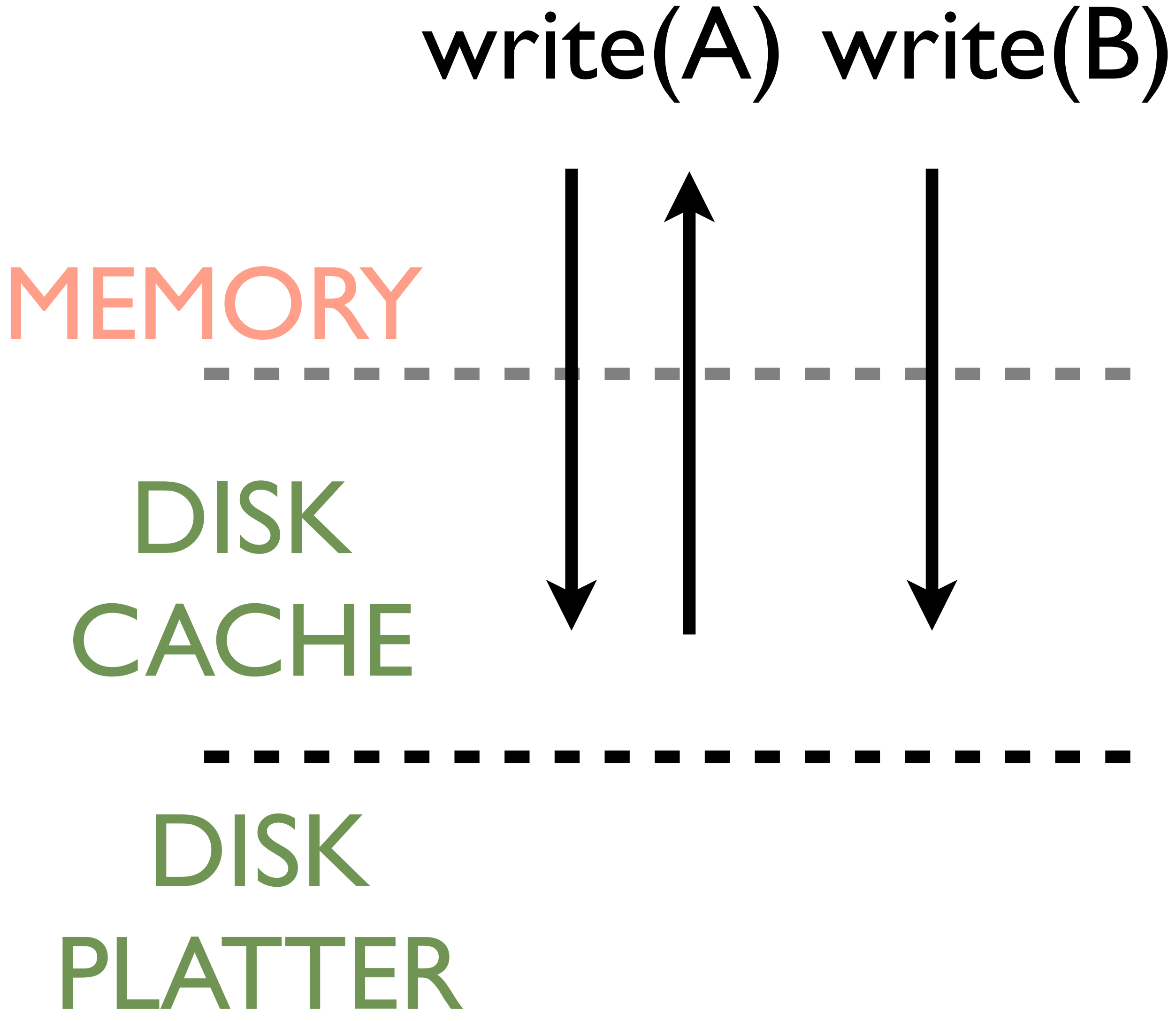
Writes are first buffered, then destaged to the non-volatile platter



Ordering Writes in Disks

Modern disk drives have on-board RAM caches

Writes are first buffered, then destaged to the non-volatile platter



write(A) write(B)

MEMORY

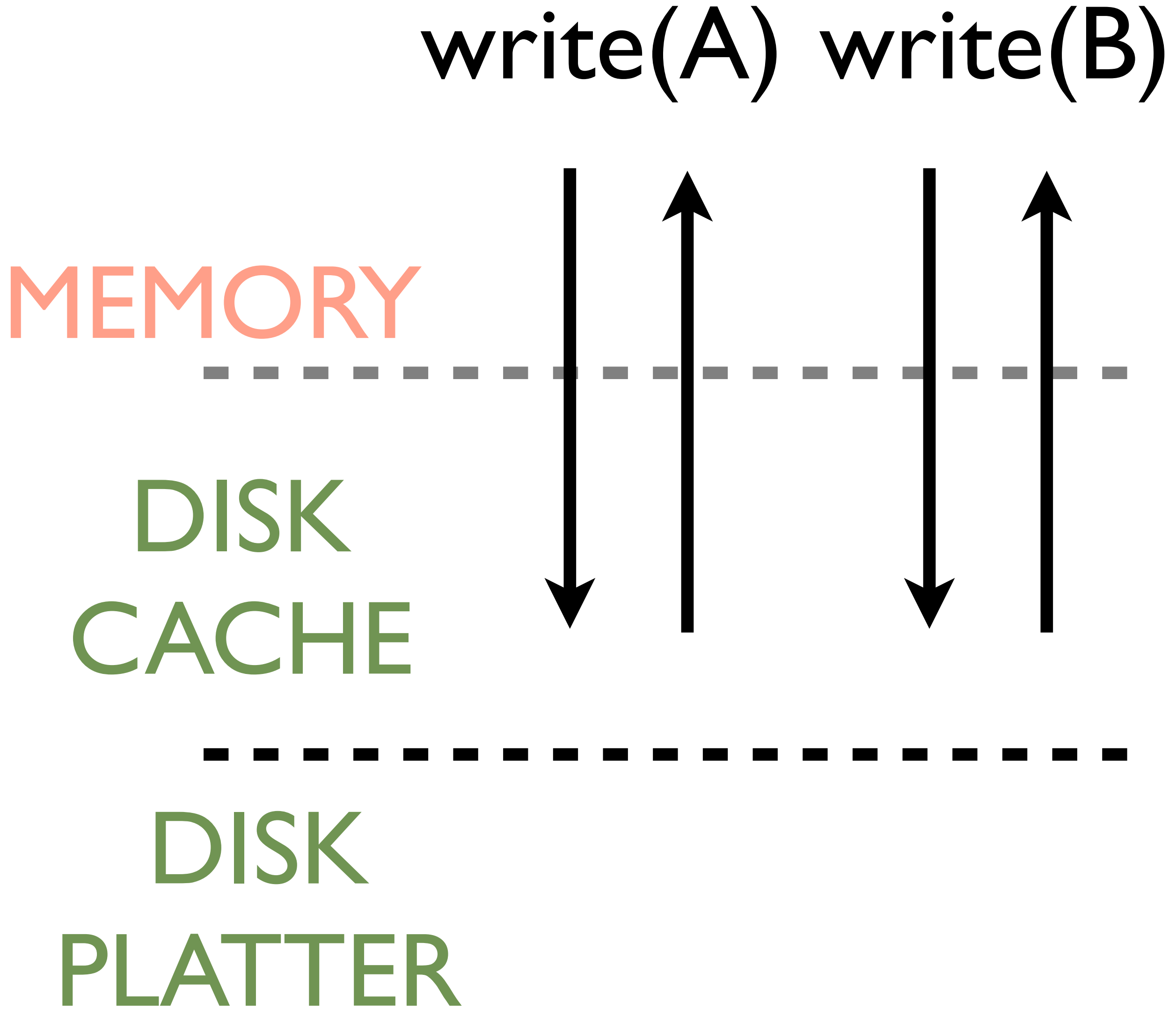
DISK CACHE

DISK PLATTER

Ordering Writes in Disks

Modern disk drives have on-board RAM caches

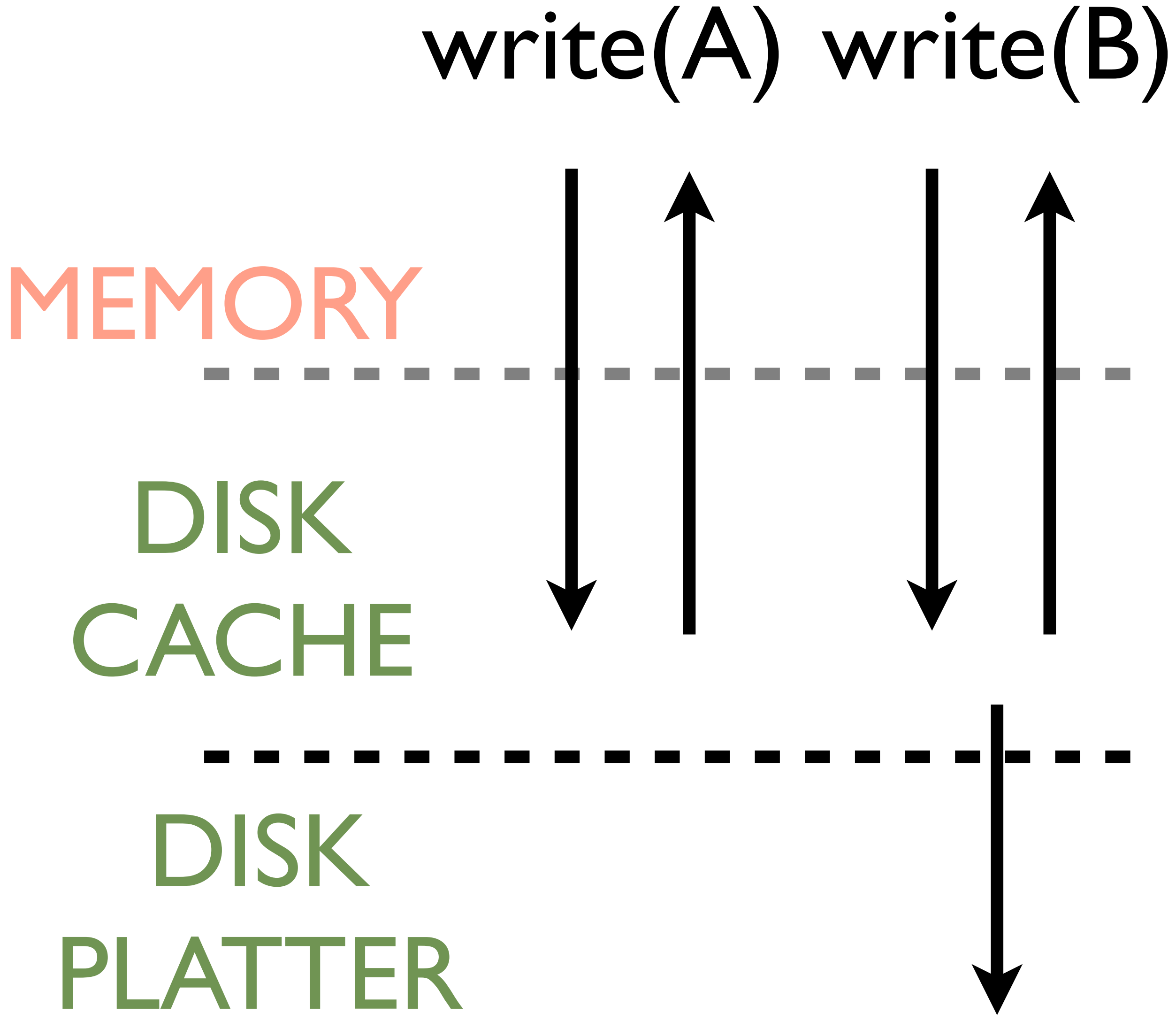
Writes are first buffered, then destaged to the non-volatile platter



Ordering Writes in Disks

Modern disk drives have on-board RAM caches

Writes are first buffered, then destaged to the non-volatile platter



write(A) write(B)

MEMORY

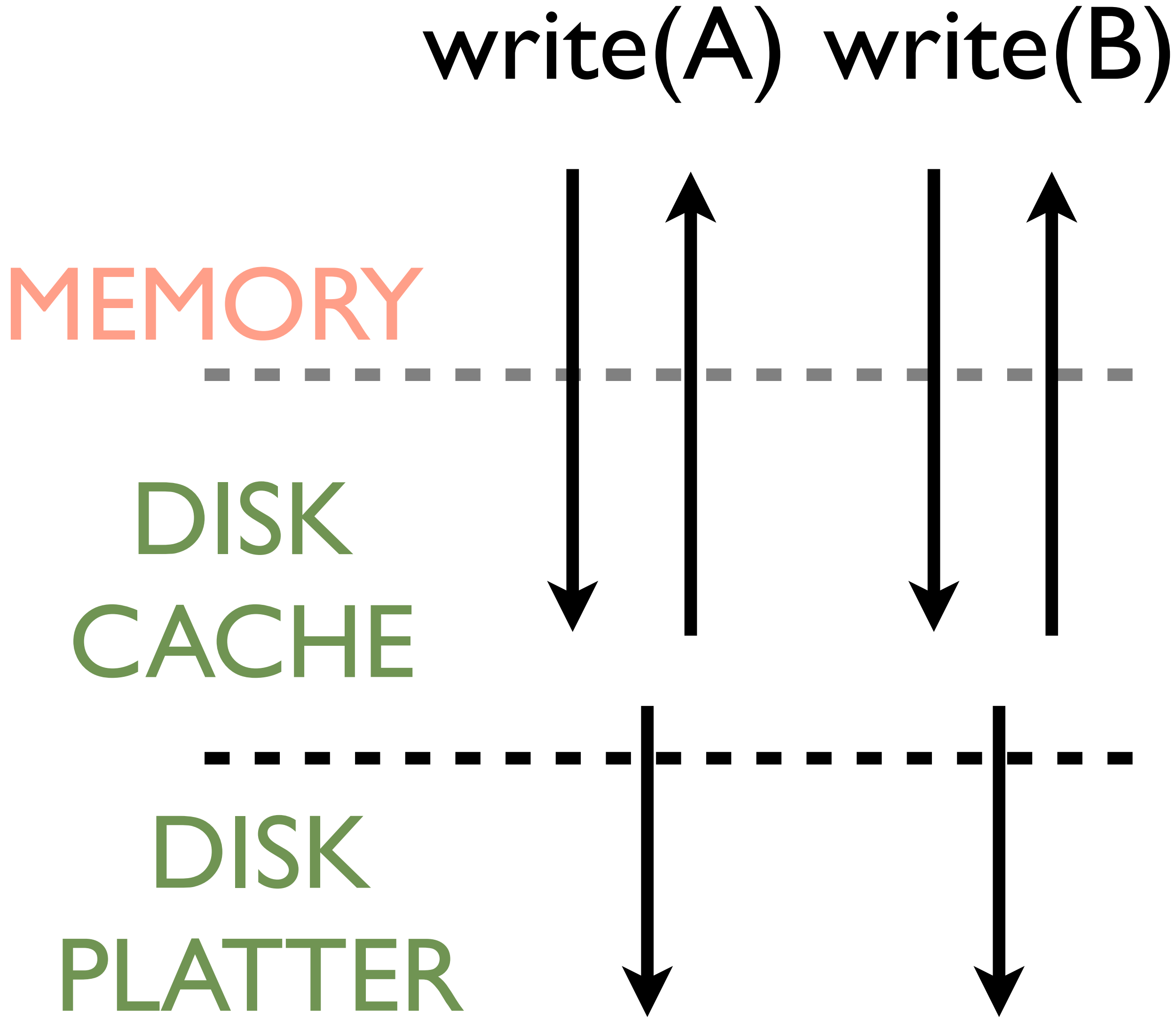
DISK CACHE

DISK PLATTER

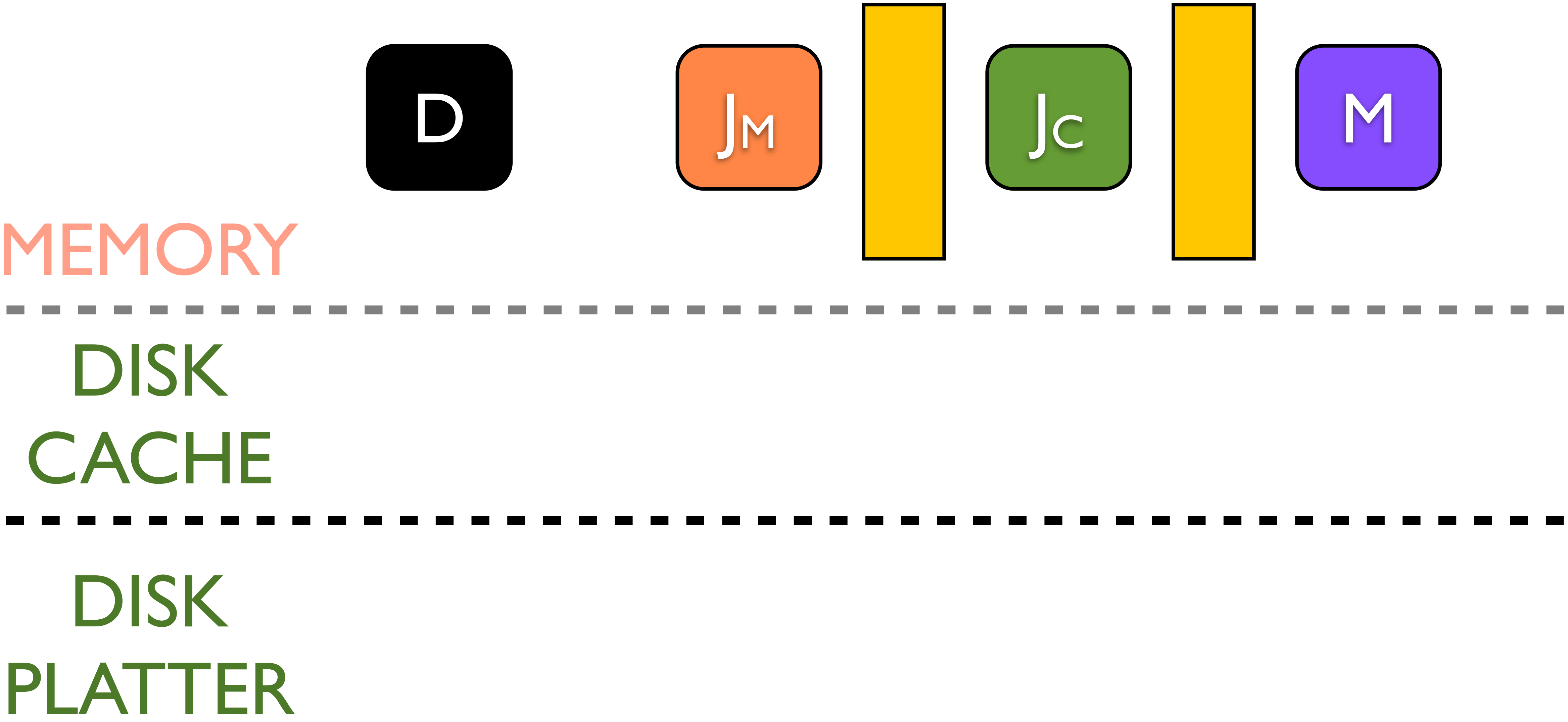
Ordering Writes in Disks

Modern disk drives have on-board RAM caches

Writes are first buffered, then destaged to the non-volatile platter



Using Flushes to Order Writes



D

J_M

J_C

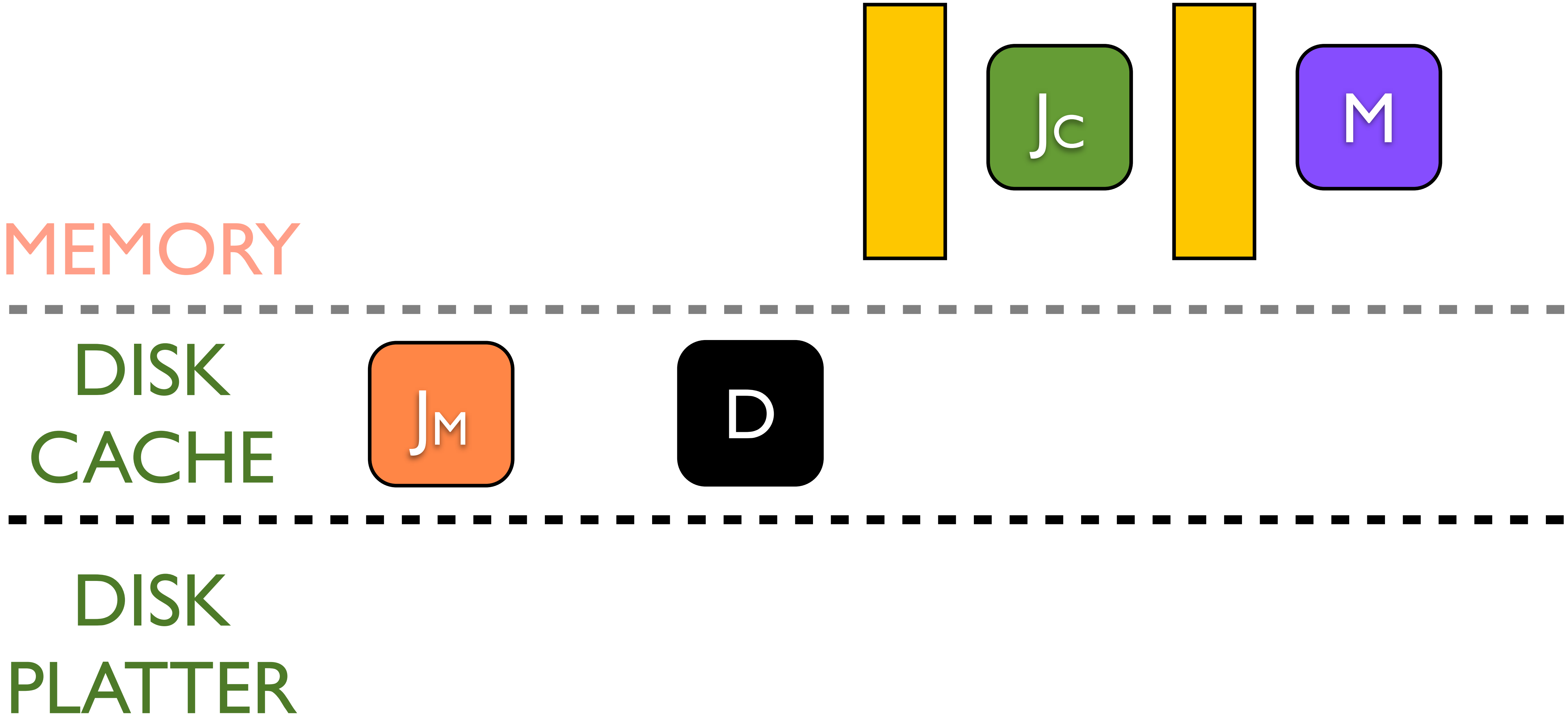
M

MEMORY

DISK CACHE

DISK PLATTER

Using Flushes to Order Writes

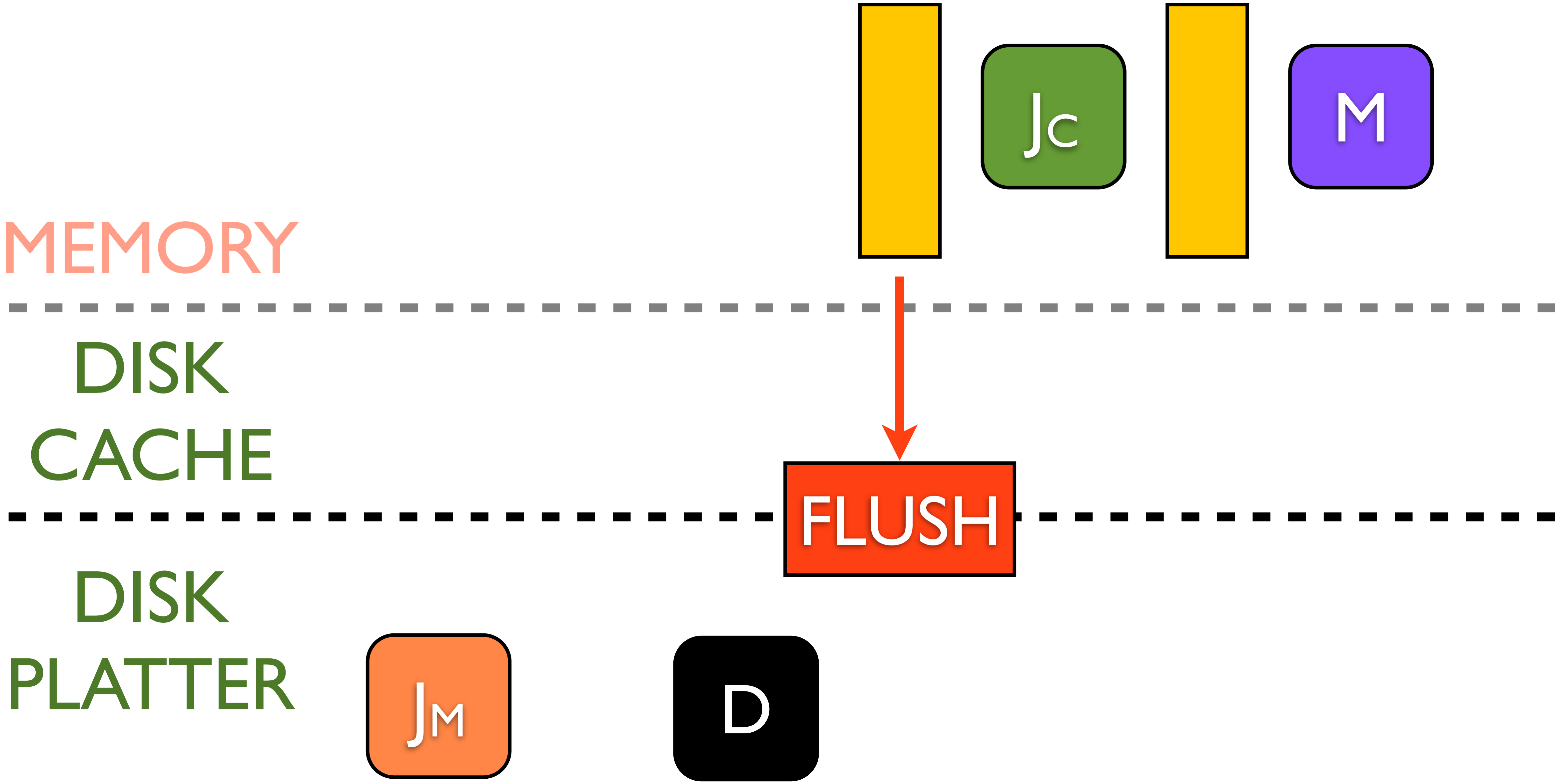


MEMORY

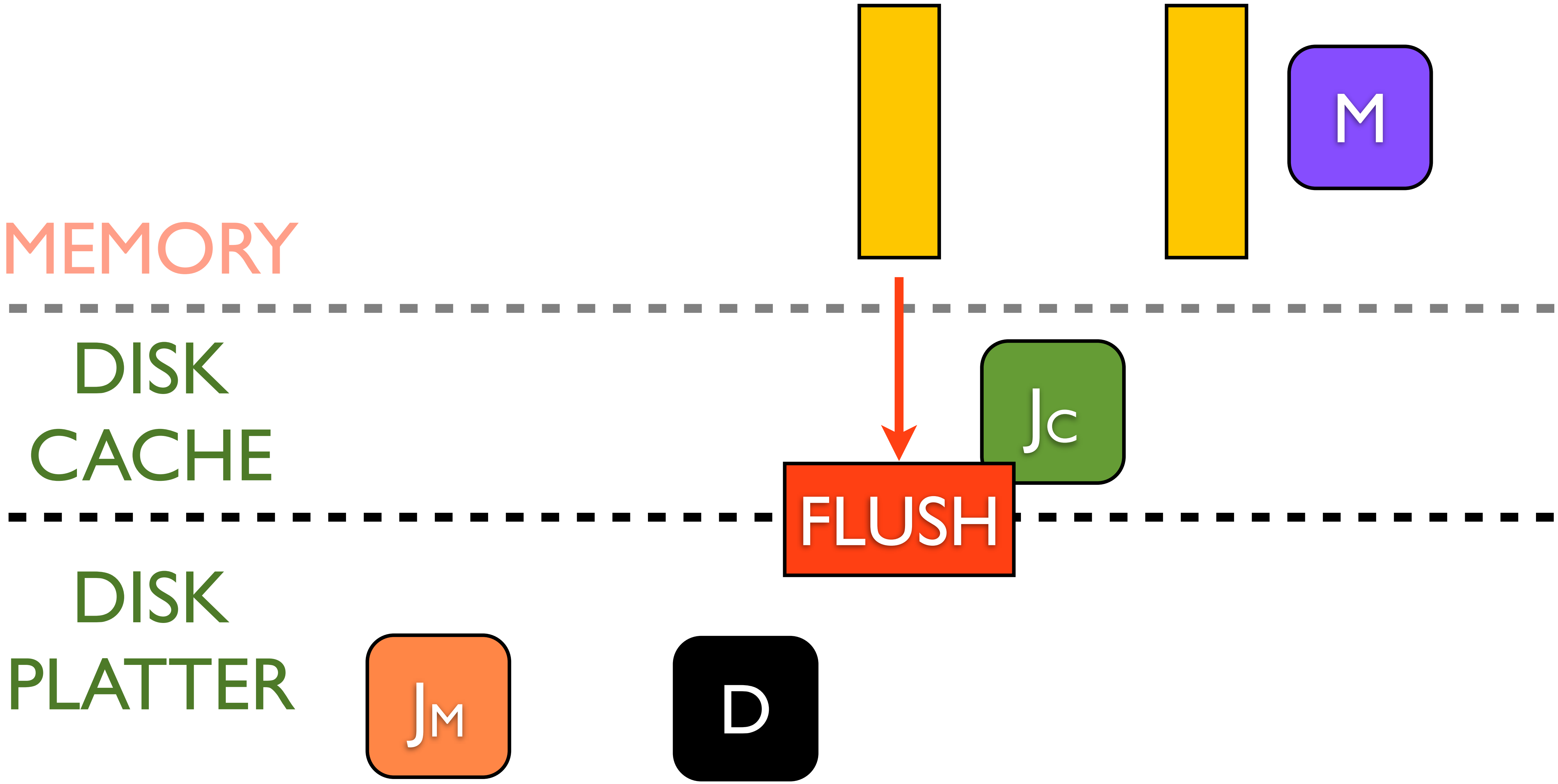
DISK CACHE

DISK PLATTER

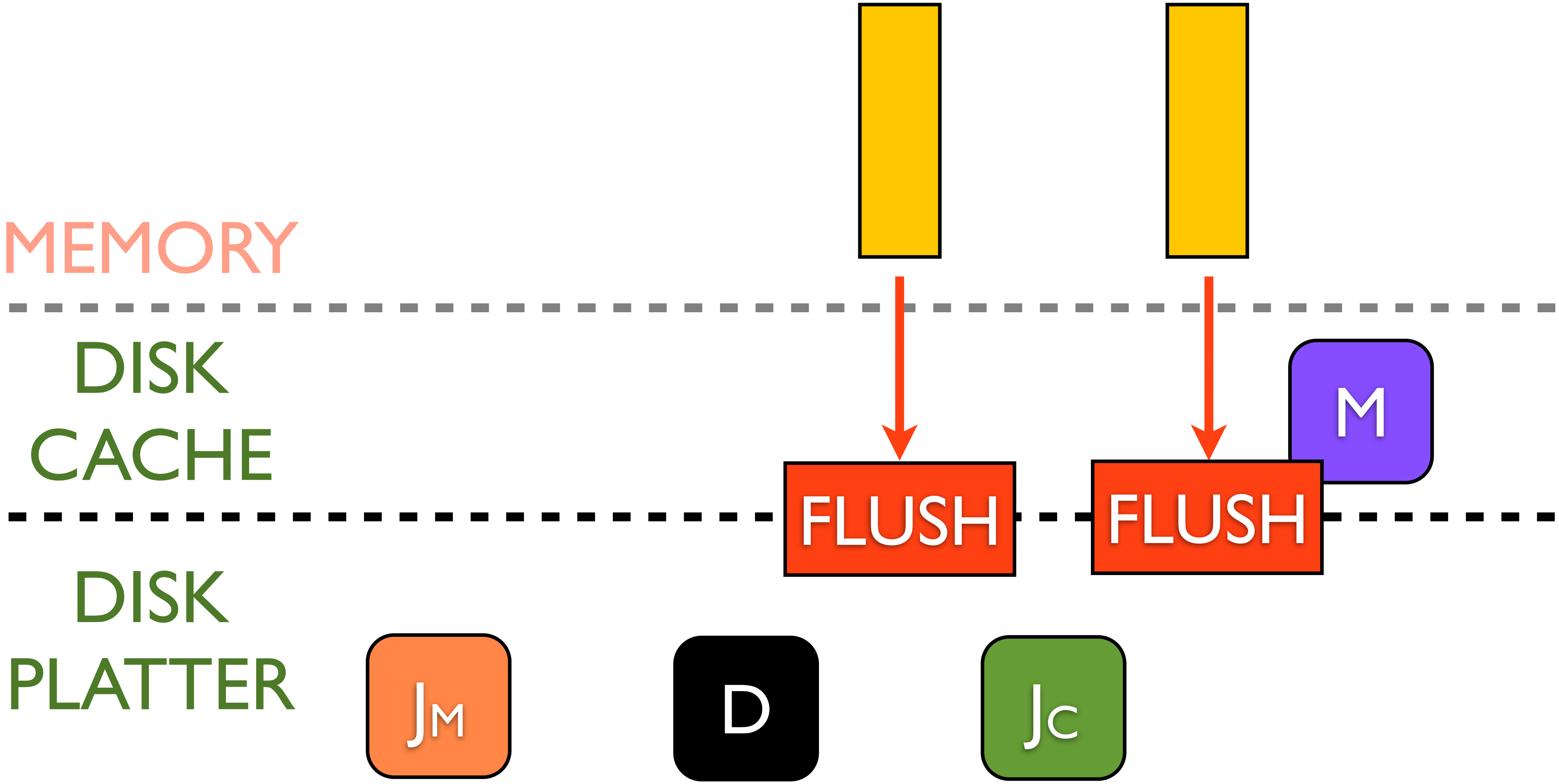
Using Flushes to Order Writes



Using Flushes to Order Writes



Using Flushes to Order Writes



Default Journaling is Pessimistic

Default Journaling is Pessimistic

Assume crash is going to happen

Default Journaling is Pessimistic

Assume crash is going to happen

Do extra work during normal runtime

Default Journaling is Pessimistic

Assume crash is going to happen

Do extra work during normal runtime

Maintain consistency using flushes

Default Journaling is Pessimistic

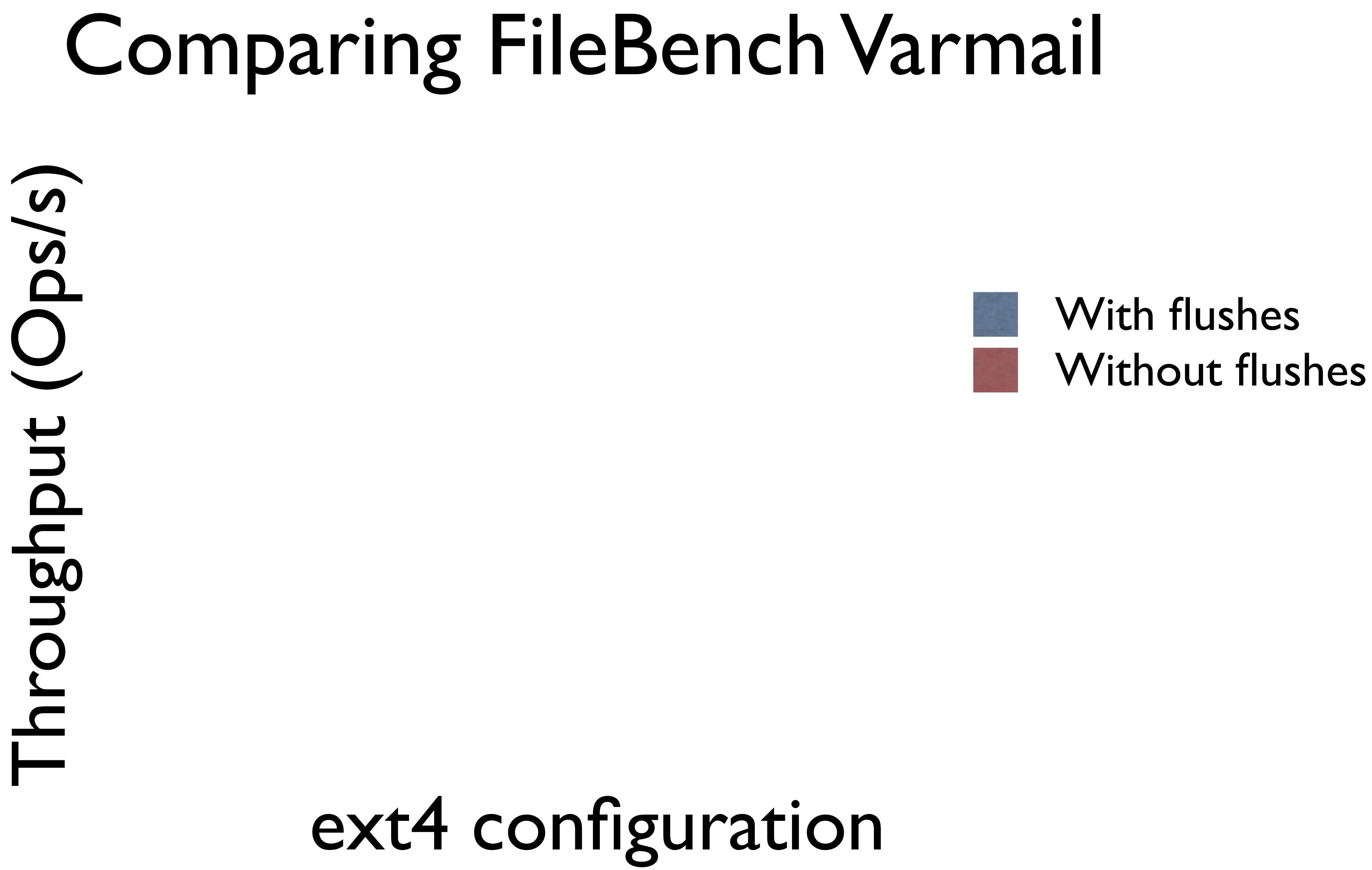
Assume crash is going to happen

Do extra work during normal runtime

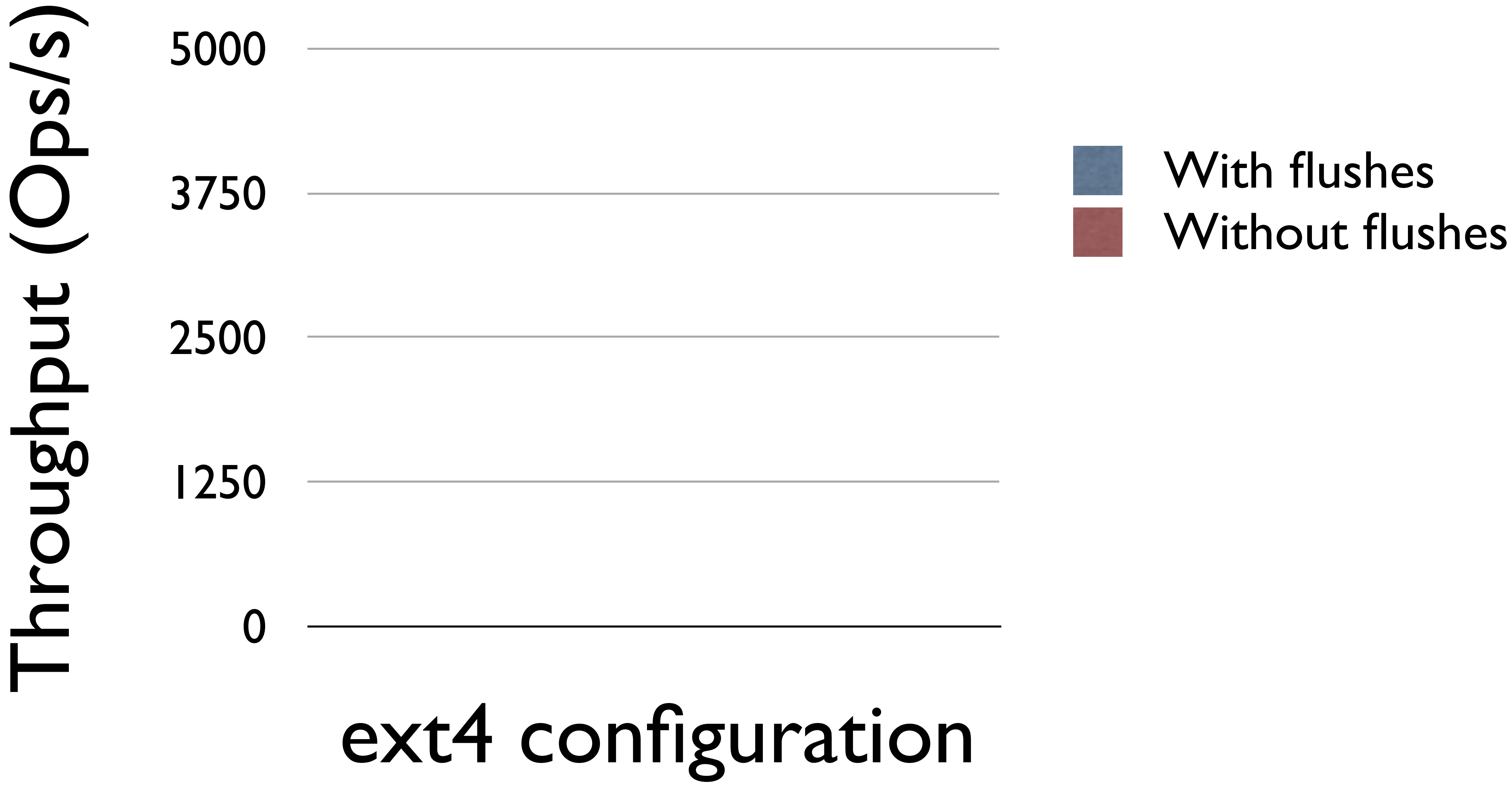
Maintain consistency using flushes

If crash does not happen, flushes are not actually needed

Flushing Performance Impact

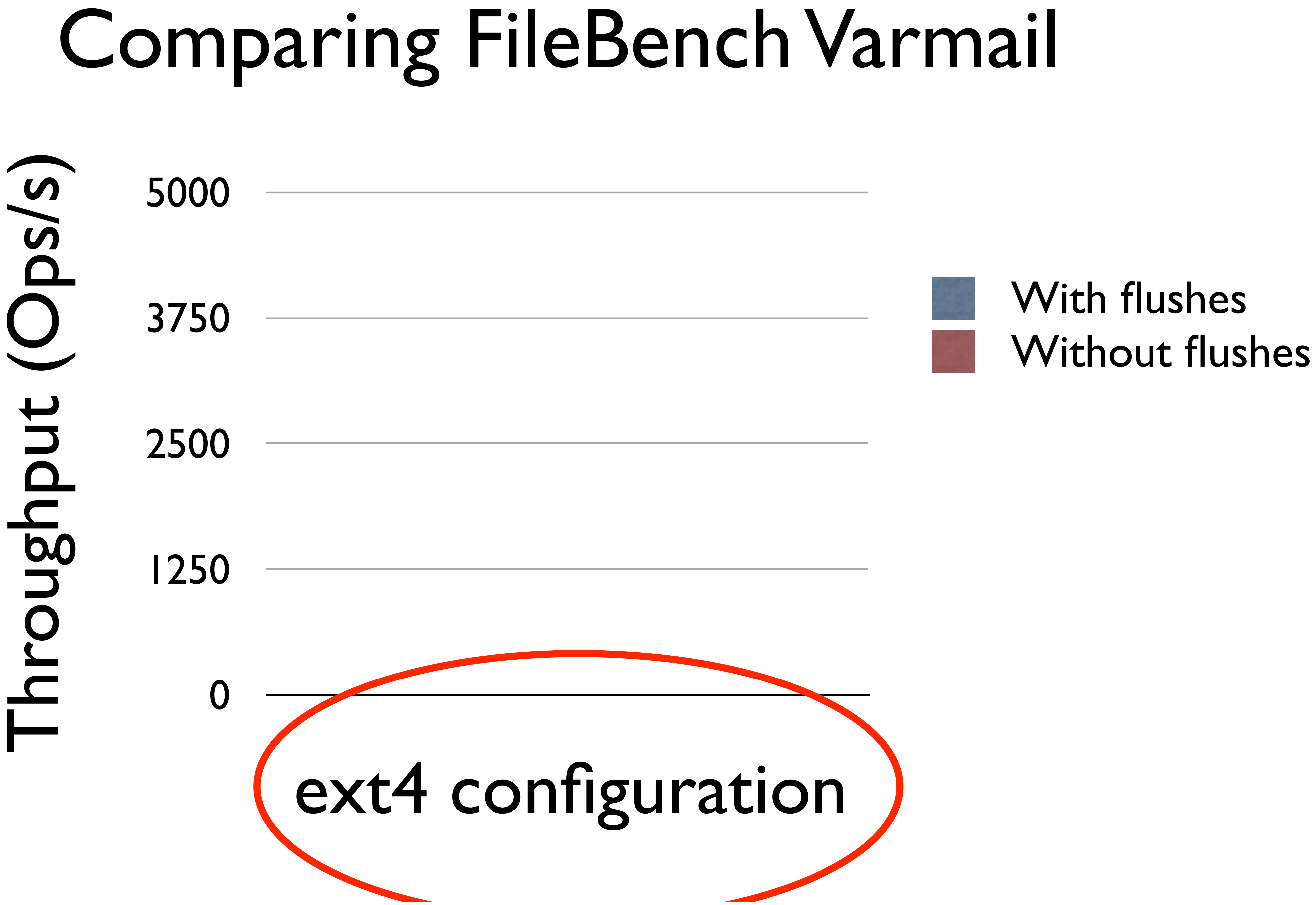

Comparing FileBench Varmail

Flushing Performance Impact

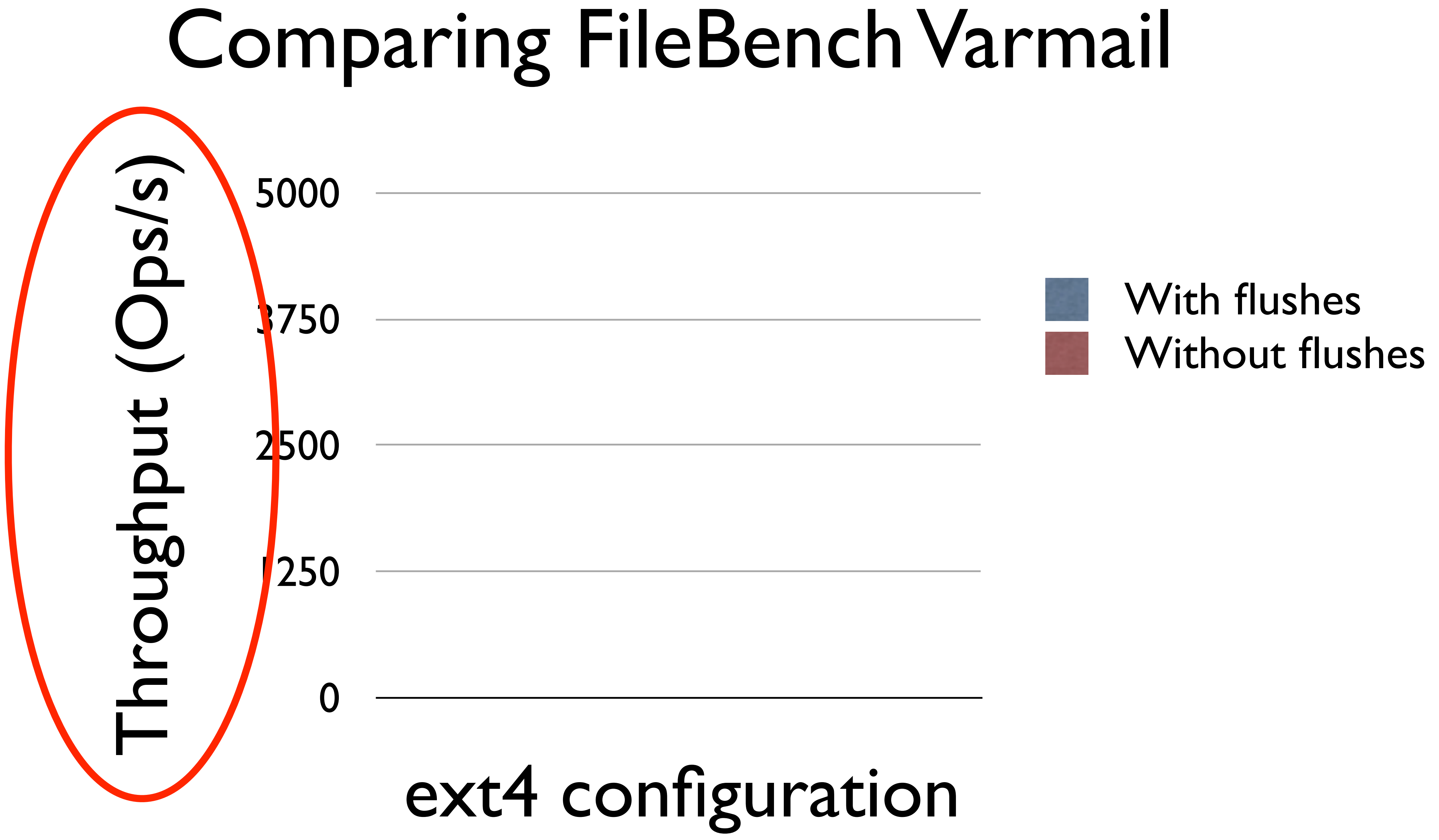


Comparing FileBench Varmail

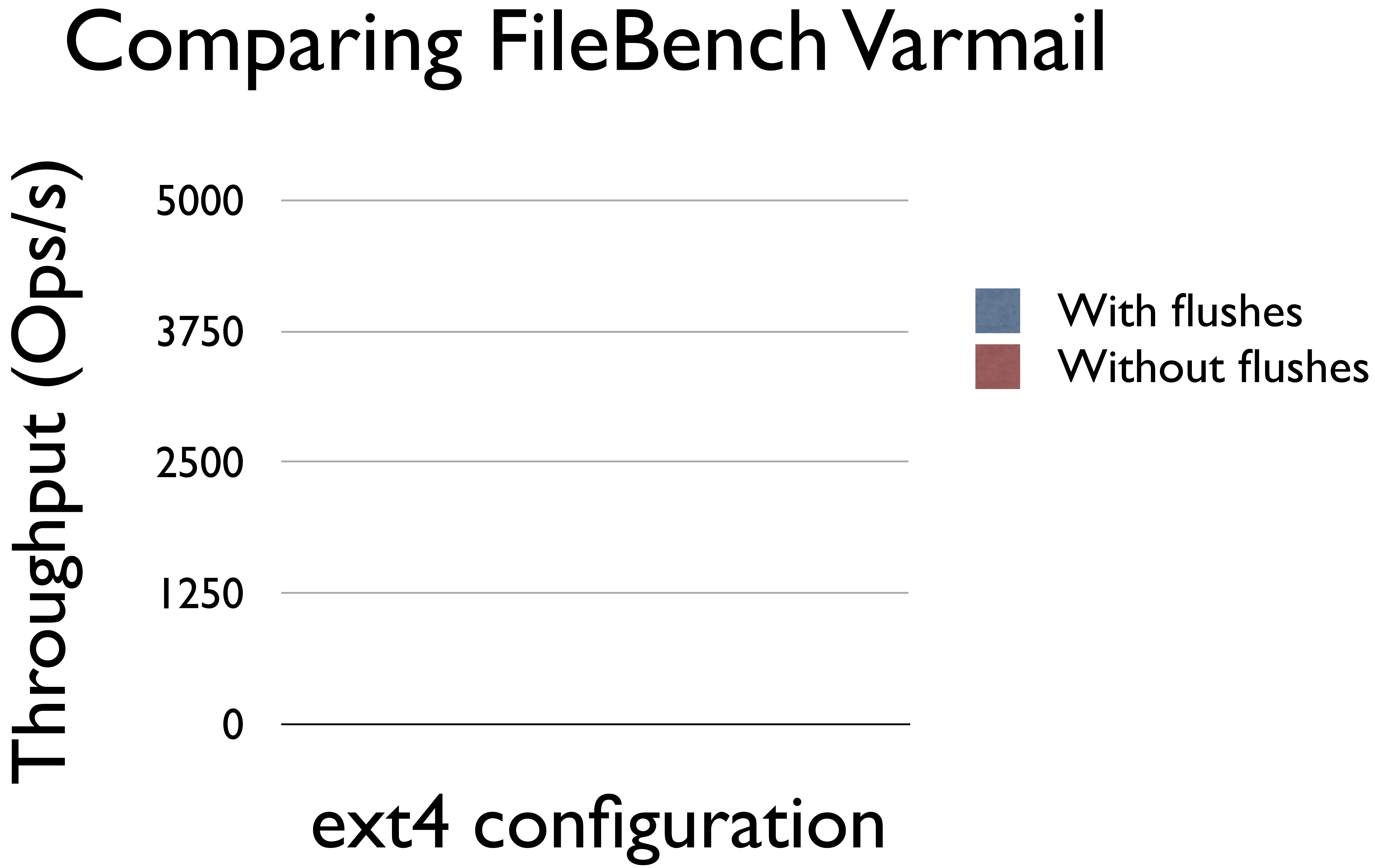
Flushing Performance Impact



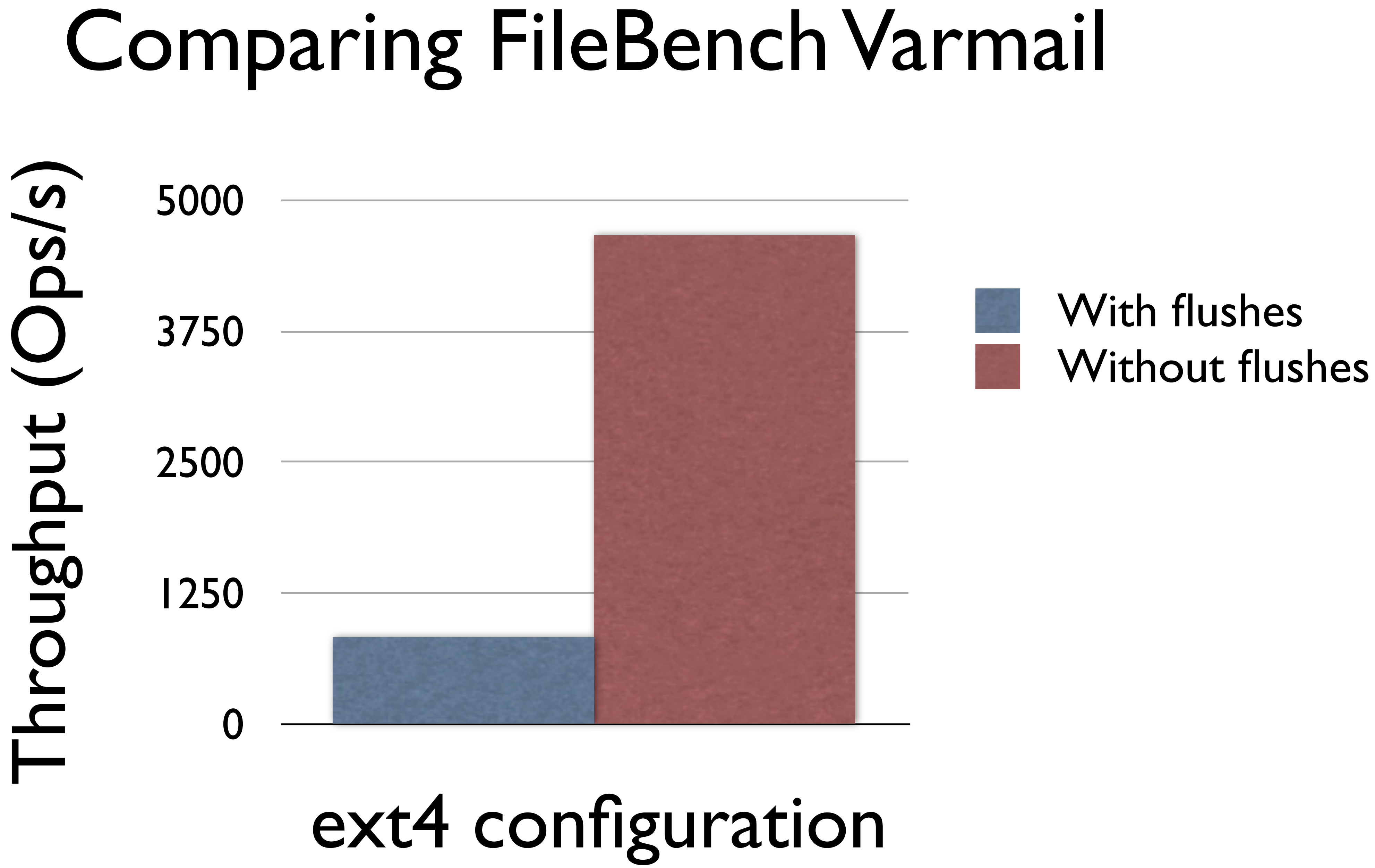
Flushing Performance Impact



Flushing Performance Impact

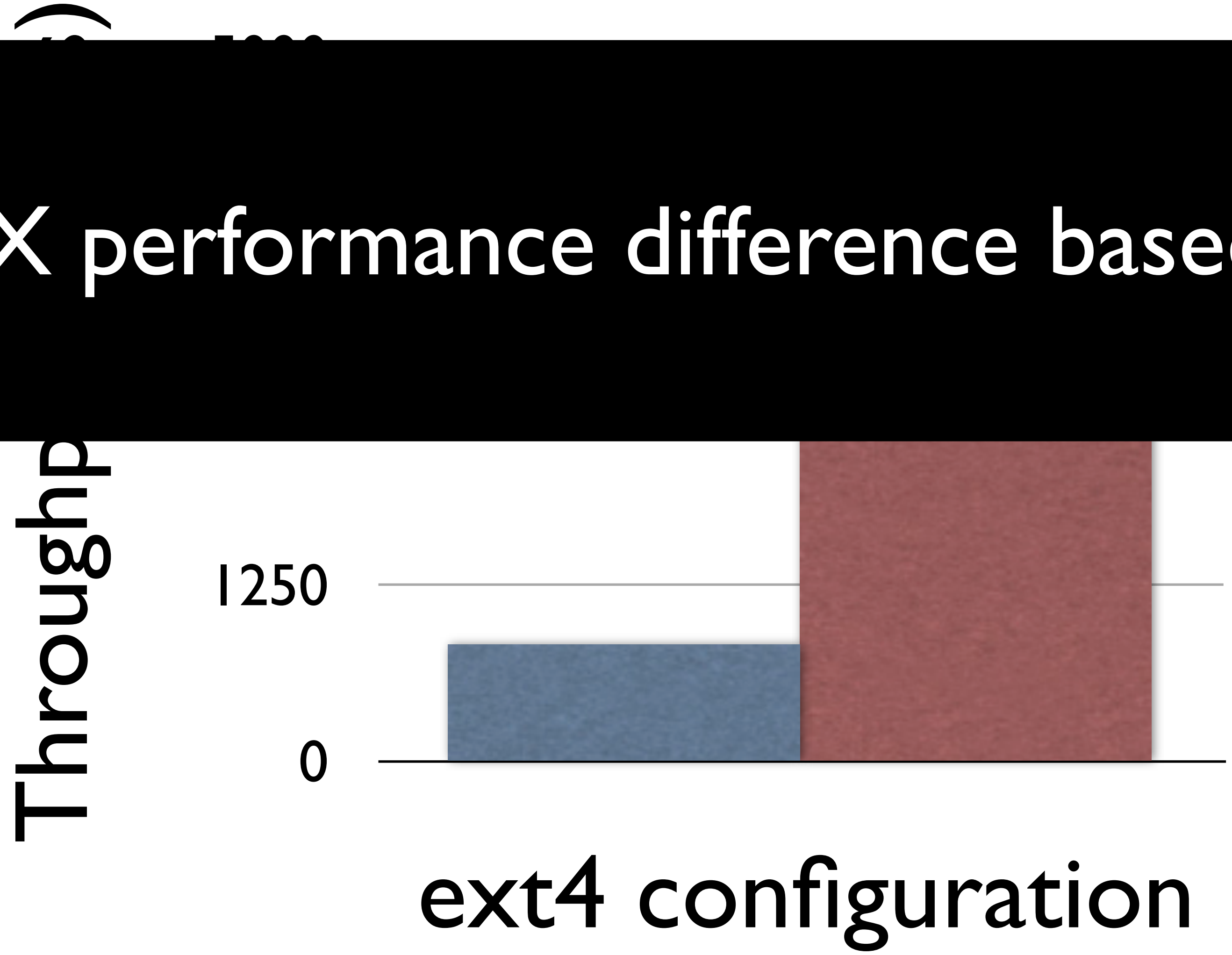


Flushing Performance Impact



Flushing Performance Impact

Comparing FileBench Varmail



~ 5X performance difference based on flushing!

Journaling
Without Flushes

Journaling Without Flushes

Journaling Without Flushes

Many practitioners turn off flushes because of performance degradation

Journaling Without Flushes

Many practitioners turn off flushes because of performance degradation

Ex: ext3 by default did not enable flushes for many years

Journaling Without Flushes

Many practitioners turn off flushes because of performance degradation

Ex: ext3 by default did not enable flushes for many years

They observe crashes do not cause inconsistency for some workloads

Journaling Without Flushes

Many practitioners turn off flushes because of performance degradation

Ex: ext3 by default did not enable flushes for many years

They observe crashes do not cause inconsistency for some workloads

The Ts'o Hypothesis

Kernel developer Ted Ts'o hypothesized on why inconsistency does not occur:

> "I suspect the real reason why we get away with it so much with ext3 is that the journal is usually contiguous on disk, hence, when you write to the journal, it's highly unlikely that commit block will be written and the blocks before the commit block have not. ... The most important reason, though, is that the blocks which are dirty don't get flushed out to disk right away!"

The Ts'o Hypothesis

Kernel developer Ted Ts'o hypothesized on why inconsistency does not occur:

> "I suspect the real reason why we get away with it so much with ext3 is that the journal is usually contiguous on disk, hence, when you write to the journal, it's highly unlikely that commit block will be written and the blocks before the commit block have not. ... The most important reason, though, is that the blocks which are dirty don't get flushed out to disk right away!"

Re-ordering does not happen due to layout and checkpointing delay

Probabilistic Crash Consistency

We set out to investigate the Ts'o hypothesis

- Given a workload, what is the risk of causing inconsistency upon crash?
- What are the factors which that contribute to the risk?

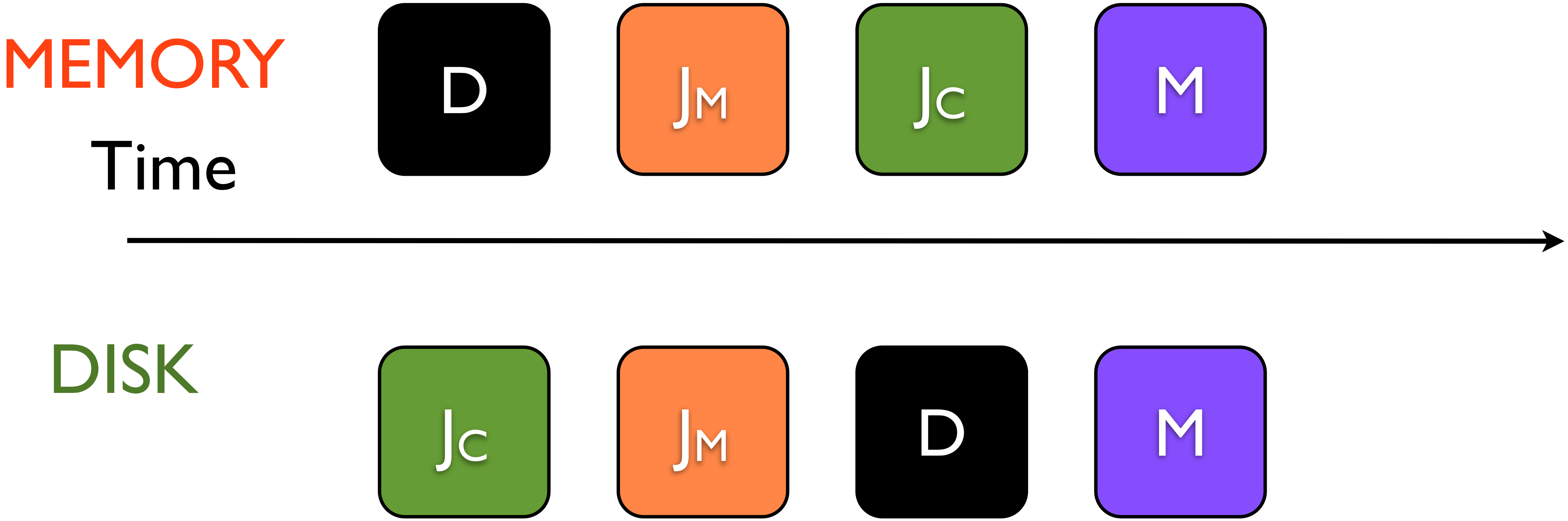
Probabilistic Crash Consistency

We ran different workloads on ext4 without flushes

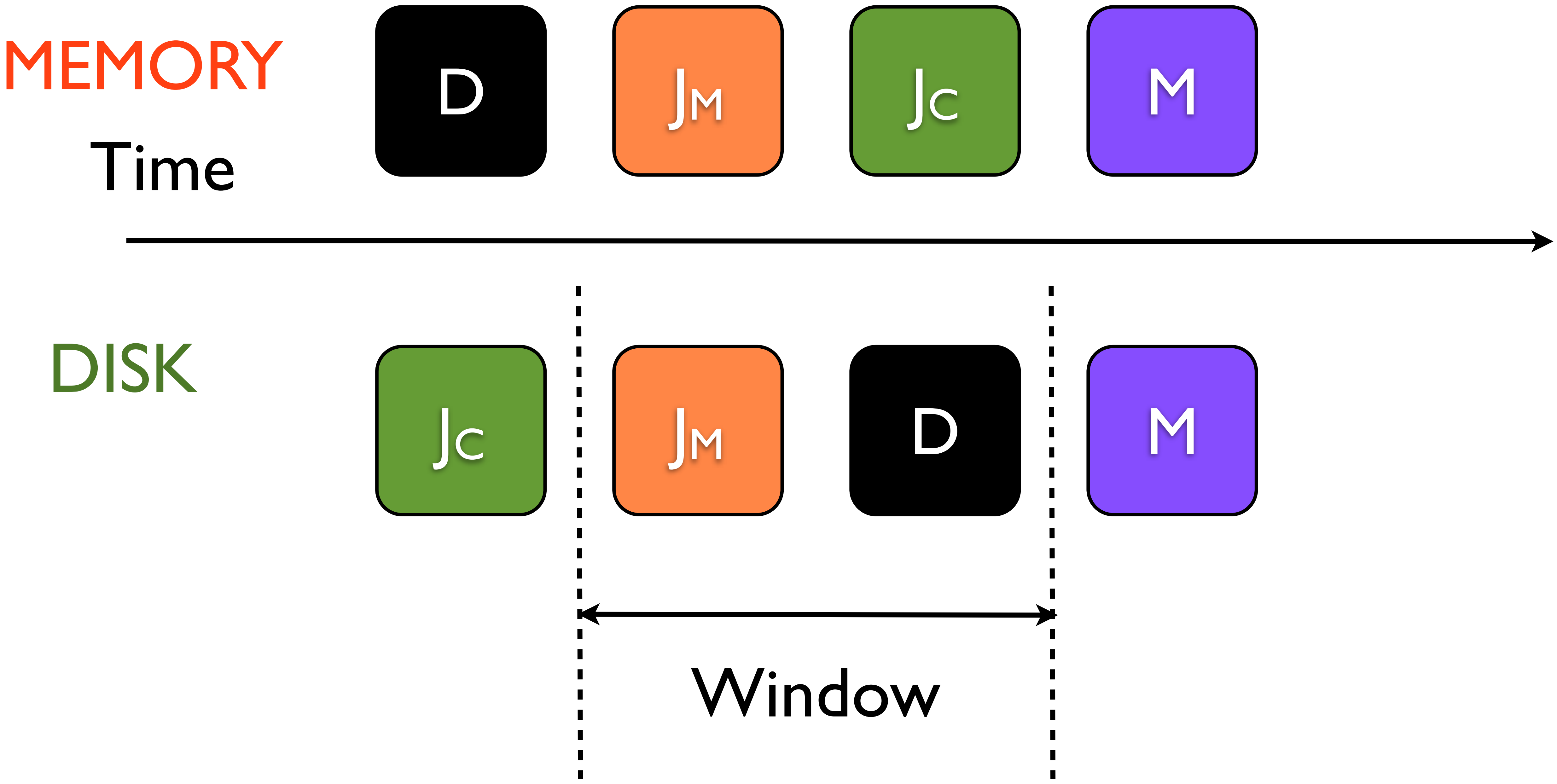
We collected the traces at the disk level

We ran them on DiskSim simulator

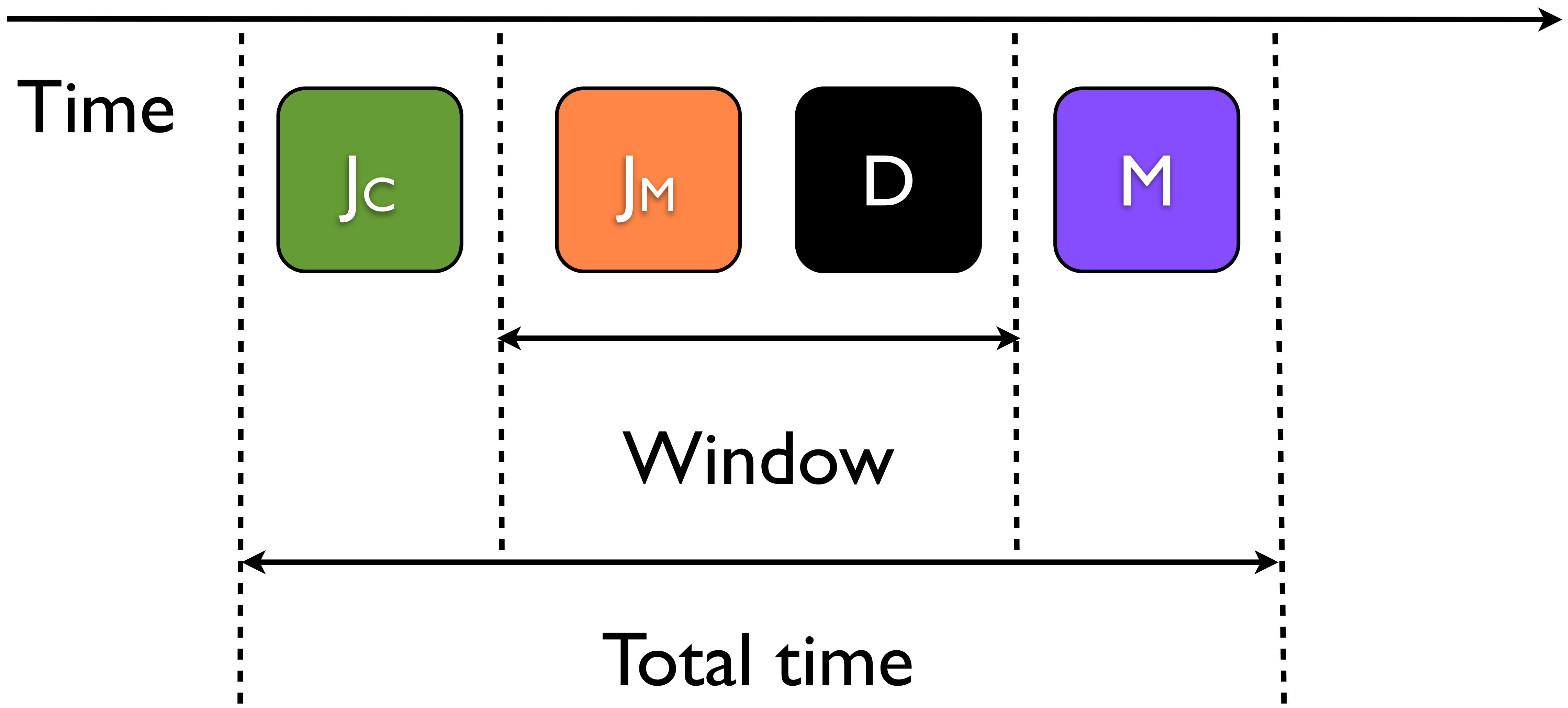
Probabilistic Crash Consistency



Probabilistic Crash Consistency



Probabilistic Crash Consistency



Time

JC

JM

D

M

Window

Total time

P-inconsistency = Total time in window(s) / Total time

Types of Re-ordering

Correct Order	D	JM	JC	M

Types of Re-ordering

Correct Order	D	JM	JC	M
Early Commit	D	JC	JM	M
	JC	D	JM	M

Types of Re-ordering

Correct Order	D	JM	JC	M
Early Commit	D	JC	JM	M
	JC	D	JM	M
Early Checkpoint	D	JM	M	JC
	D	M	JM	JC
	M	D	JM	JC

Types of Re-ordering

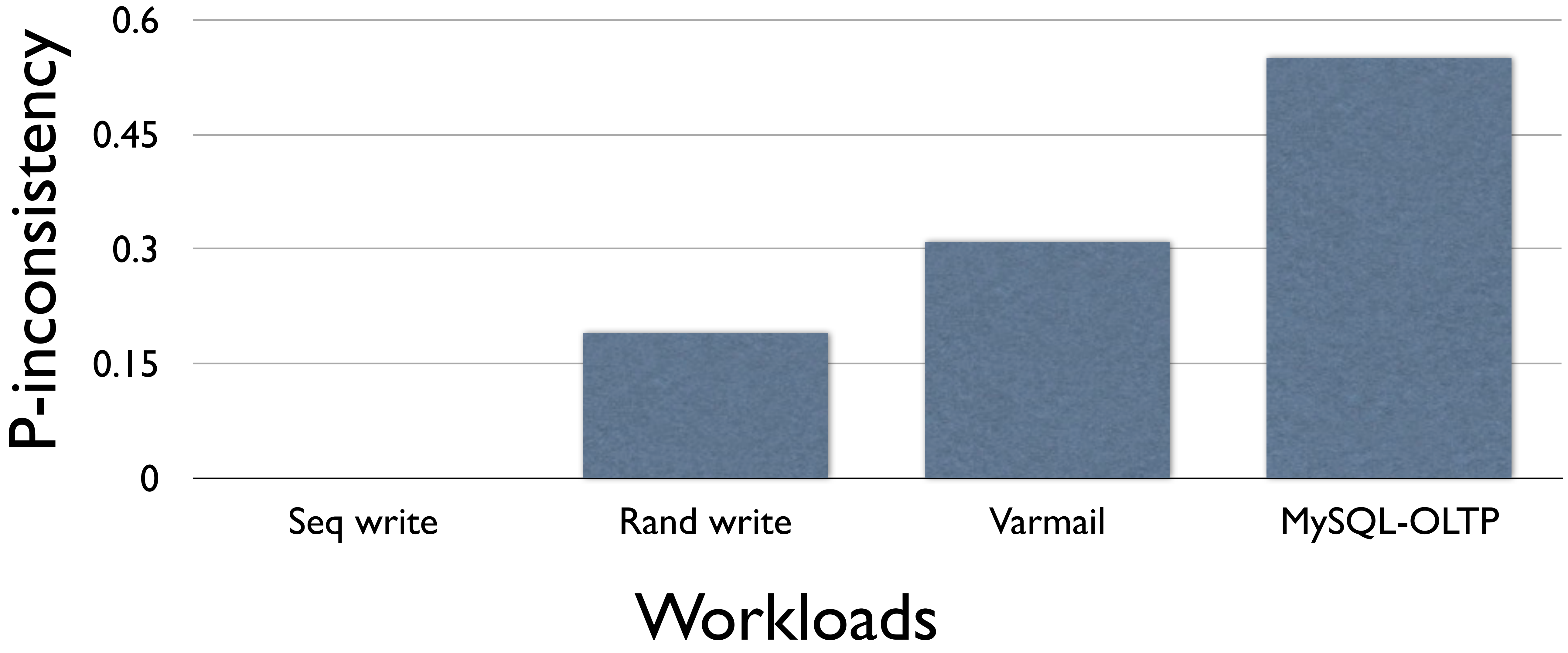
Correct Order	D	J_M	J_C	M
Early Commit	D J_C	J_C D	J_M J_M	M M
Early Checkpoint	D D M	J_M M D	M J_M J_M	J_C J_C J_C
Transaction Misorder		J_{Ci}	J_{Ci-1}	

Probabilistic Crash Consistency

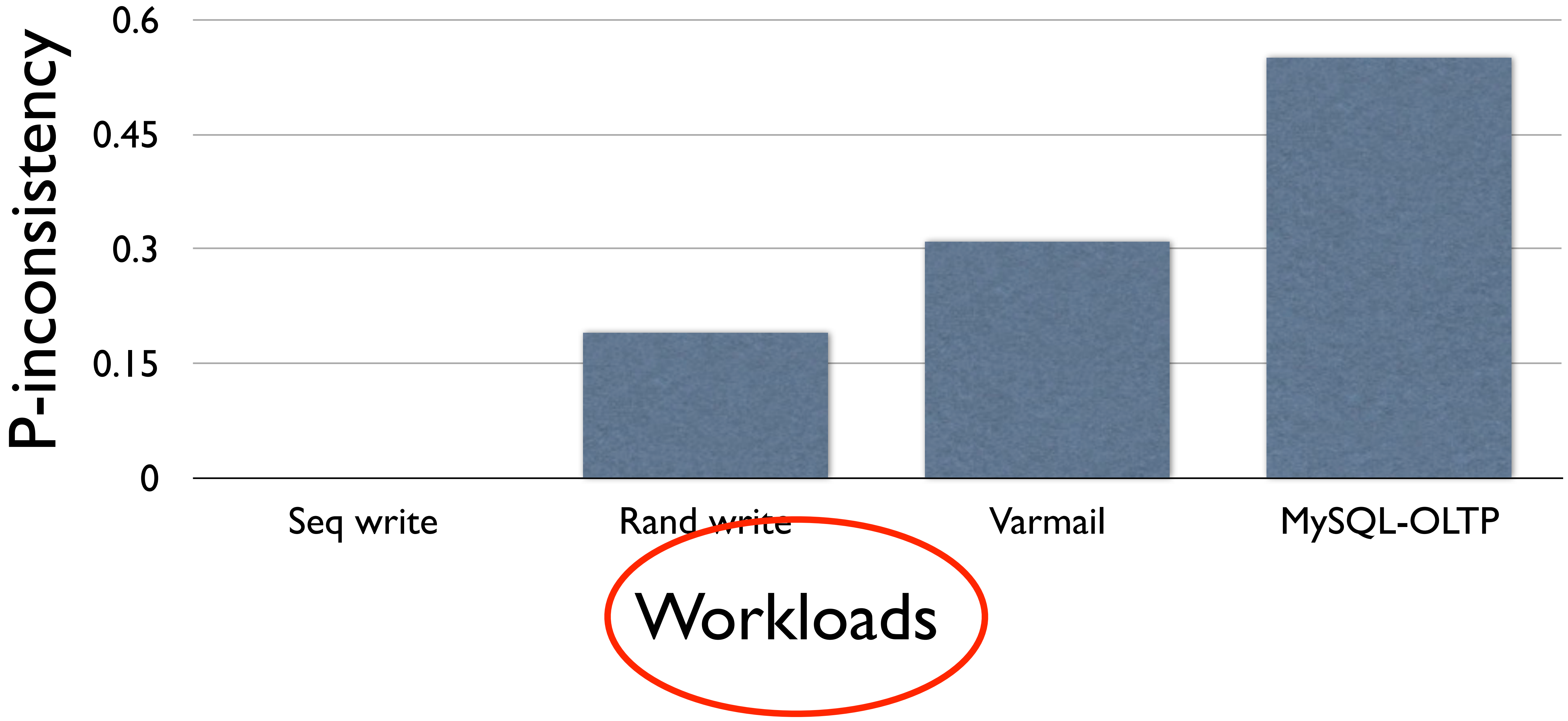
We analyzed different workloads using this framework

Calculated p-inconsistency and investigated the factors contributing to p-inconsistency

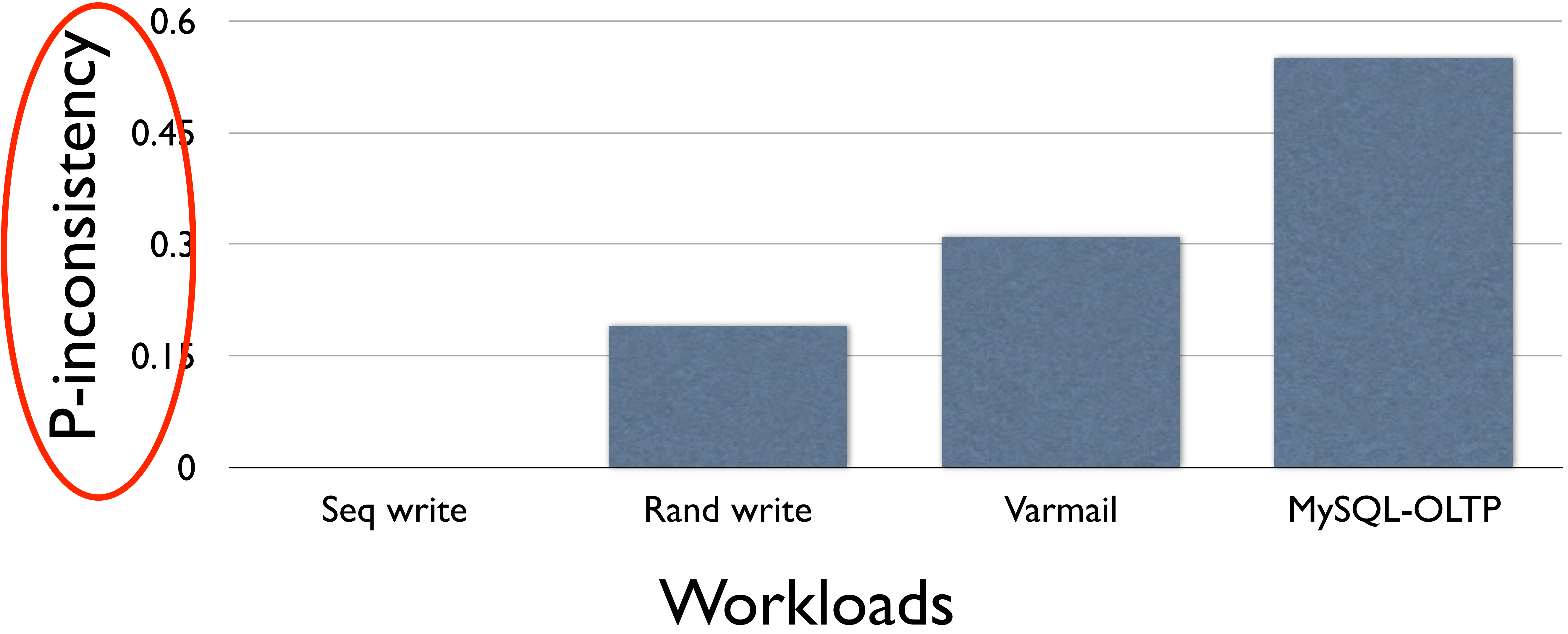
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

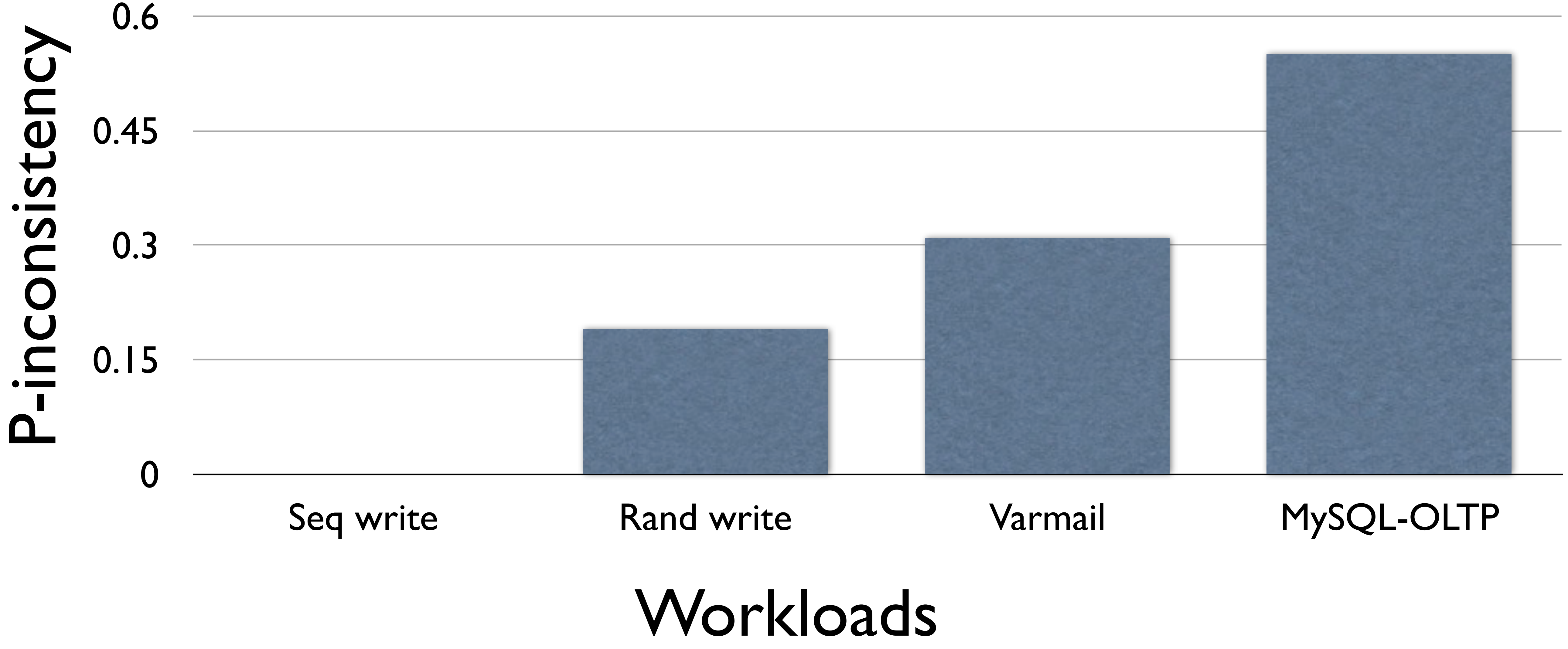
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

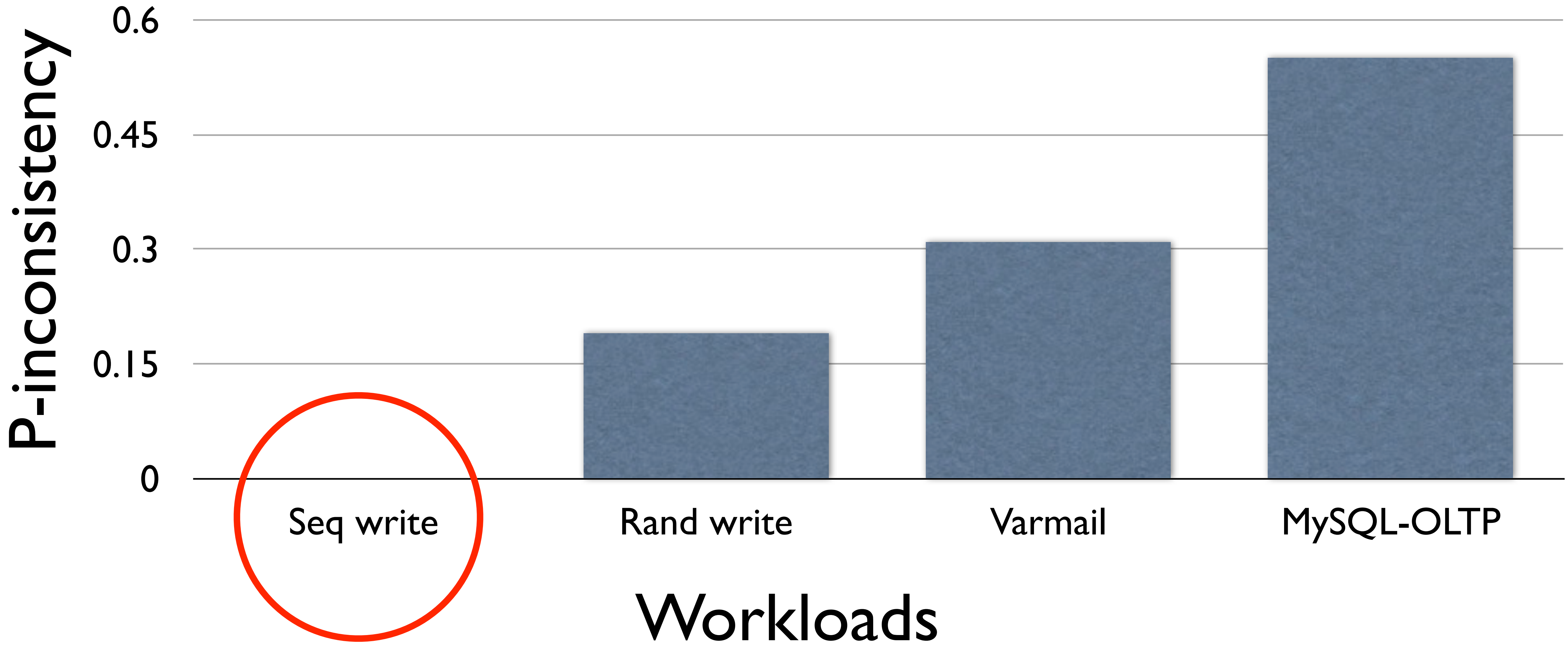
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

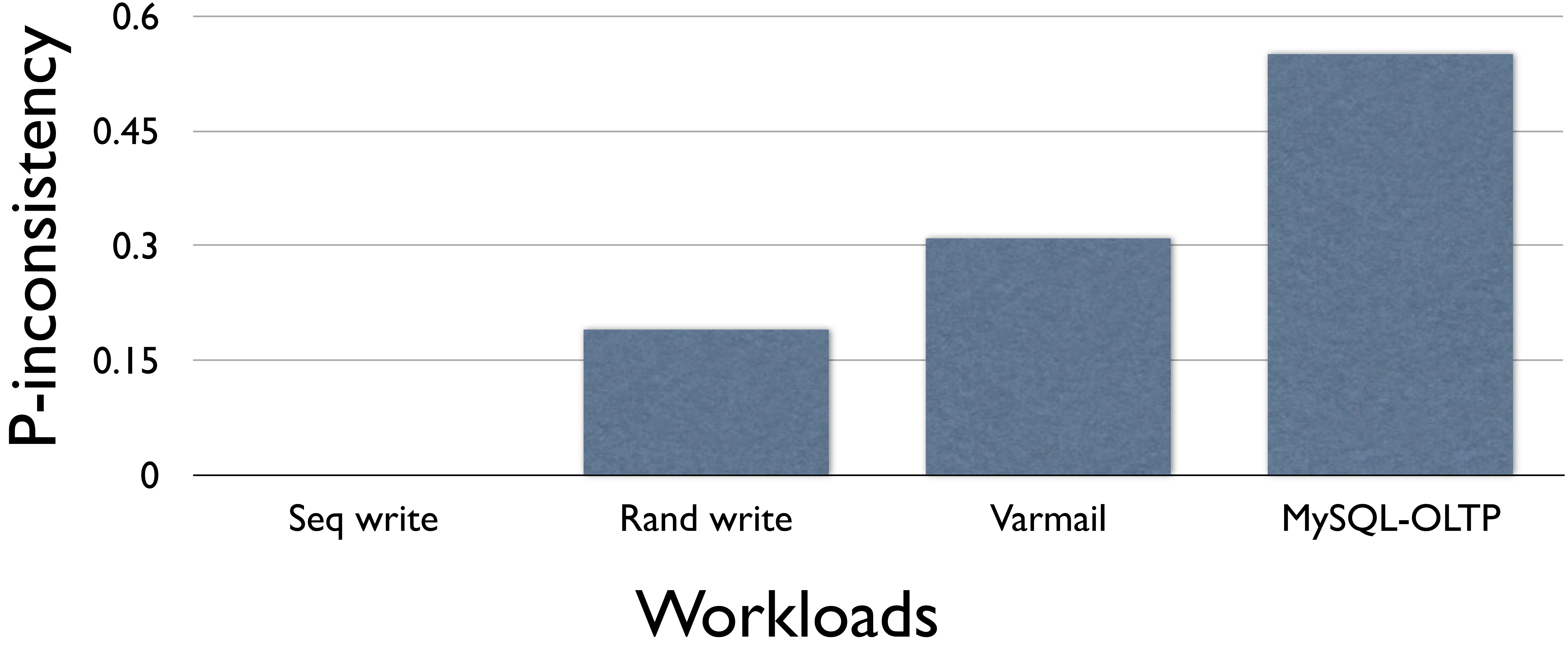
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

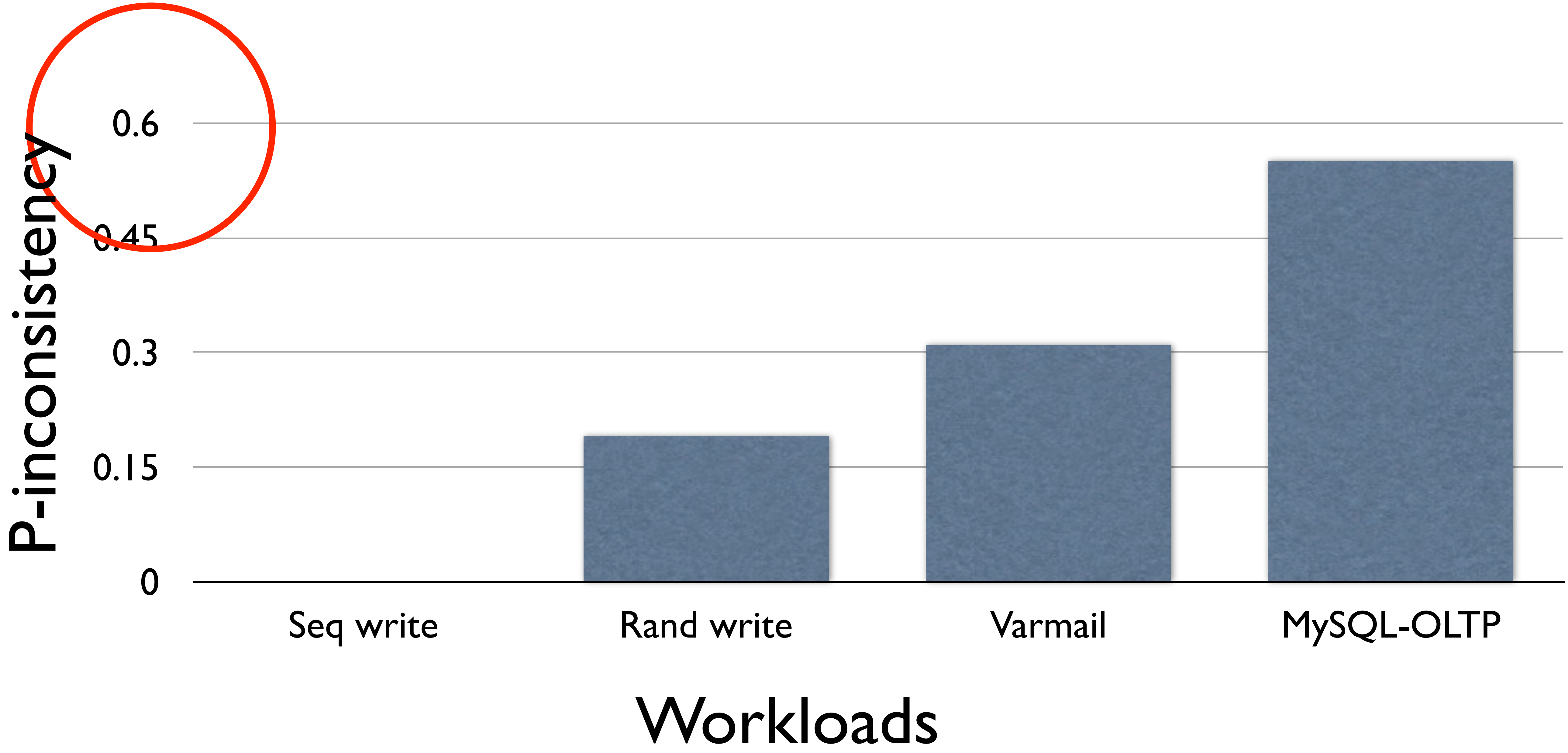
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

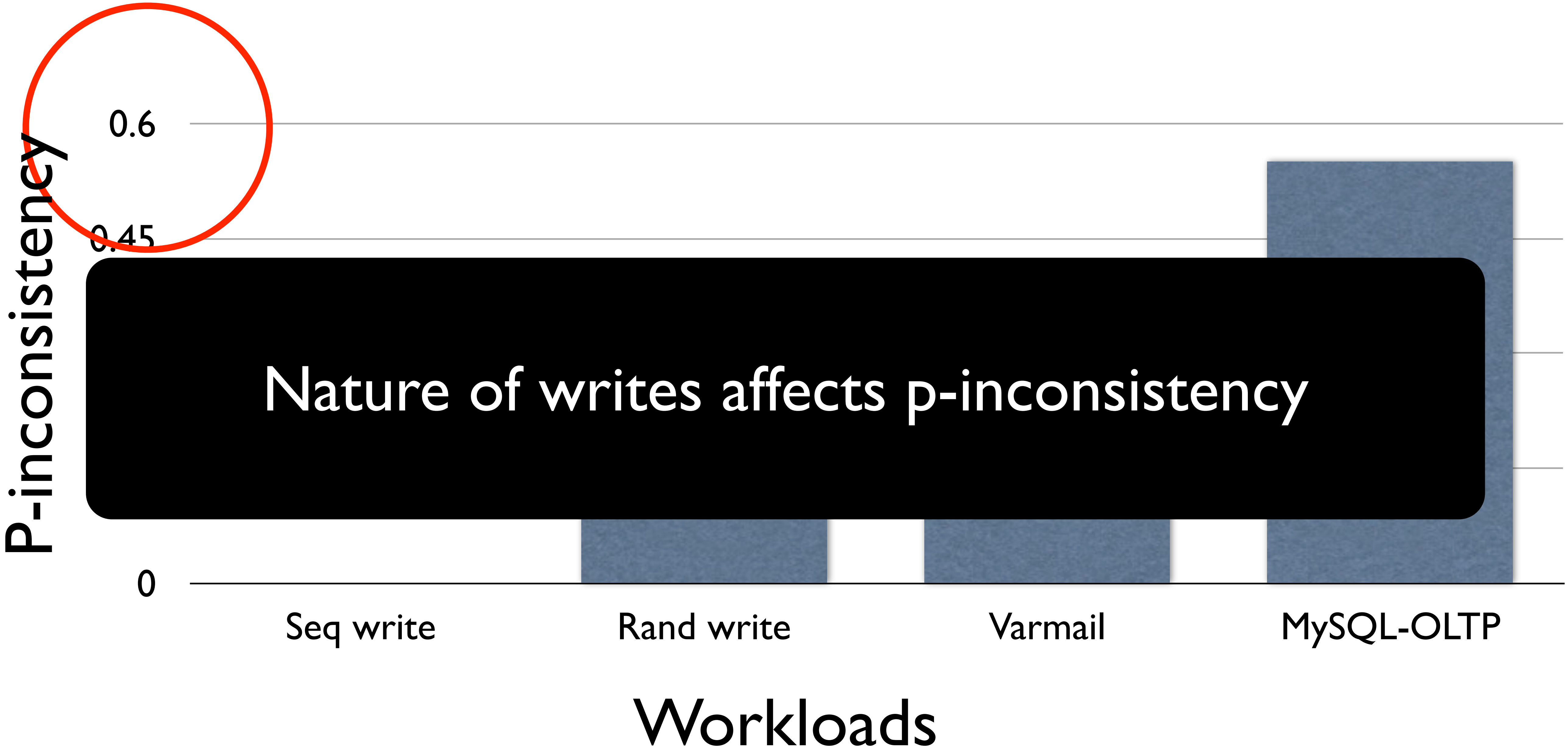
Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

Results


P-inconsistency
0.6
0.45
0.3
0.15
0
Seq write
Rand write
Varmail
MySQL-OLTP
Workloads

Results

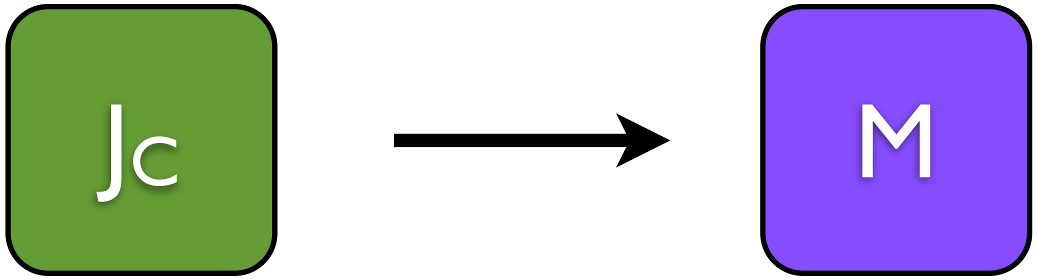


Nature of writes affects p-inconsistency

Some orderings hold in practice without flushes

Some orderings hold in practice without flushes

Checkpoint related re-orderings occurred very rarely in the workloads



- Due to the delay (~5-30 s) between committing and checkpointing a transaction

Some orderings hold in practice without flushes

If we extend that to all orderings, we get consistency without flushes

Optimistic Crash Consistency

Optimistic Crash Consistency

Optimistic commit protocol that provides consistency without flushes

Why “optimistic”?

Assume that crashes rarely happen

Eliminate flushes from runtime code

When crash happens, recover using appropriate mechanisms

Trade “freshness” for performance

Some data may be lost on a crash

Freshness

Freshness

Another aspect of crash consistency

Freshness

Another aspect of crash consistency

After a crash, what consistent state does the system recover to?

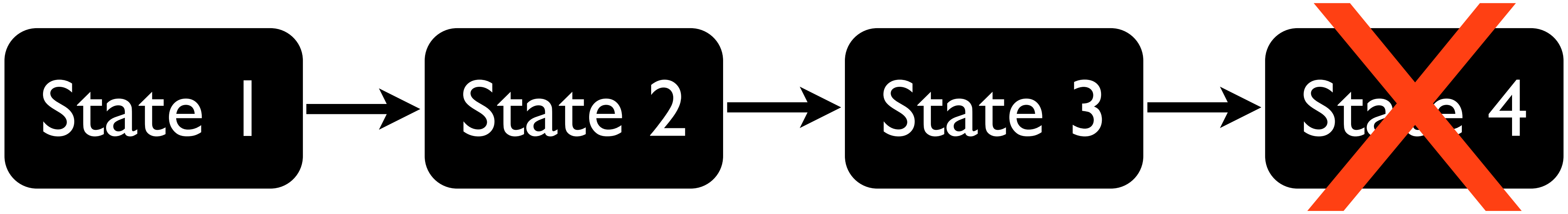
- An empty file system is consistent

Freshness

Another aspect of crash consistency

After a crash, what consistent state does the system recover to?

- An empty file system is consistent

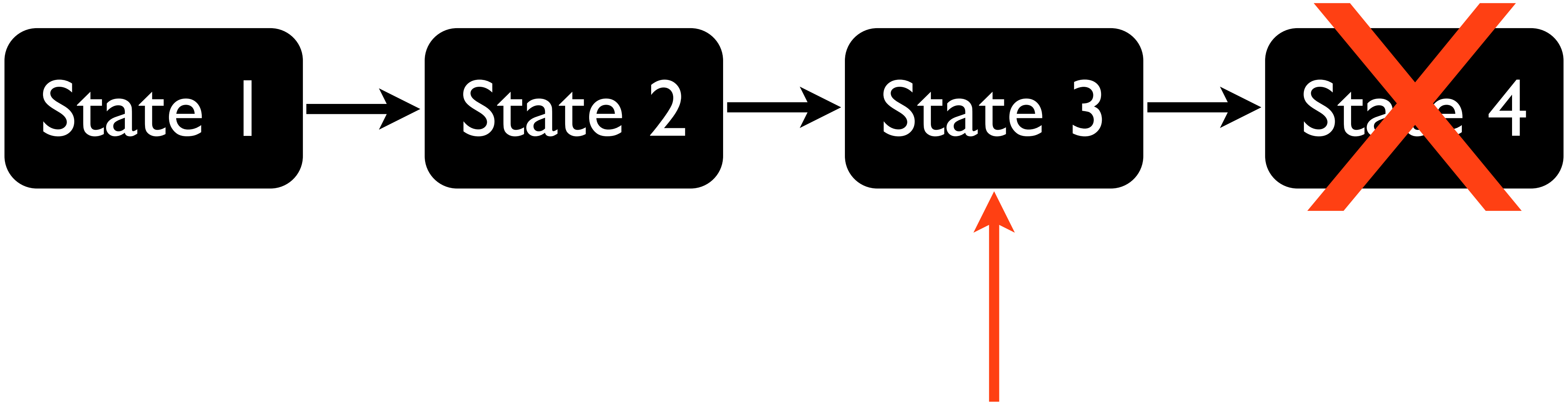


Freshness

Another aspect of crash consistency

After a crash, what consistent state does the system recover to?

- An empty file system is consistent

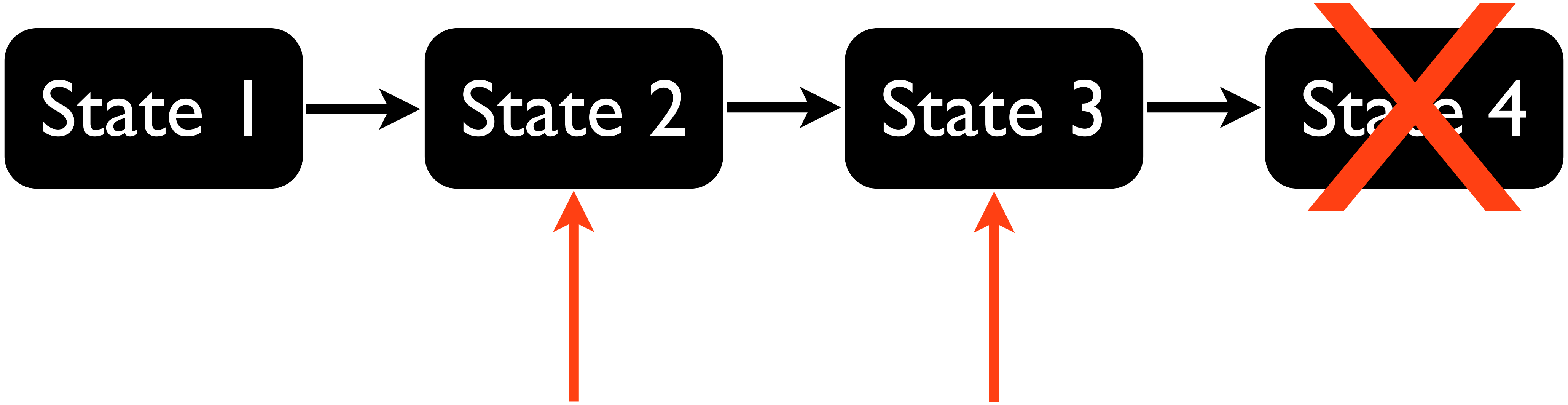


Freshness

Another aspect of crash consistency

After a crash, what consistent state does the system recover to?

- An empty file system is consistent



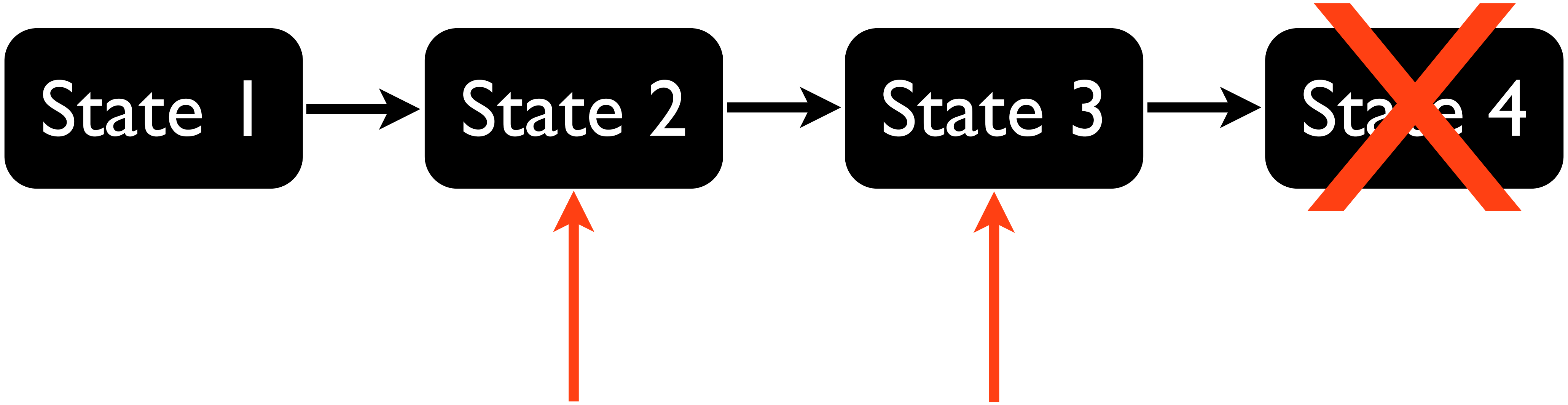
Freshness

Another aspect of crash consistency

After a crash, what consistent state does the system recover to?

- An empty file system is consistent

Many applications can tolerate stale but consistent data [Keeton04, Cipar12]



Optimistic Crash Consistency

Optimistic Crash Consistency

We design optimistic techniques to eliminate flushes in the common case

Optimistic Crash Consistency

We design optimistic techniques to eliminate flushes in the common case

It changes the ACID model: only eventual durability is provided

Optimistic Crash Consistency

We design optimistic techniques to eliminate flushes in the common case

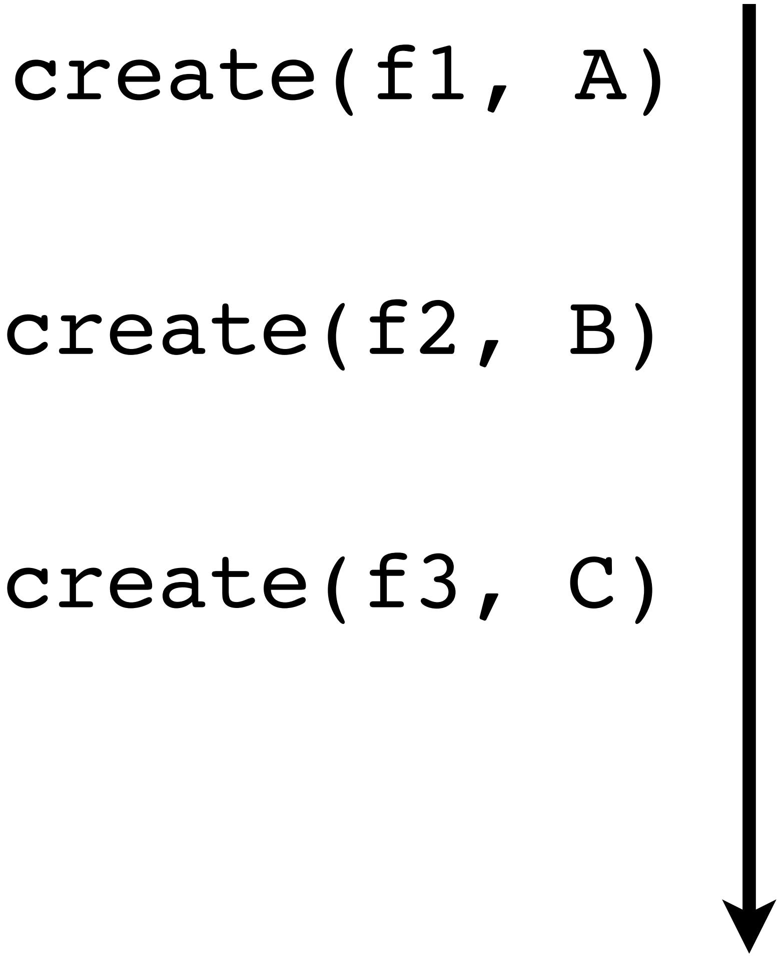
It changes the ACID model: only eventual durability is provided

We split the `fsync()` imperative into two:

- `osync()` for ordering
- `dsync()` for durability

nosync, fsync and osync

nosync, fsync and osync

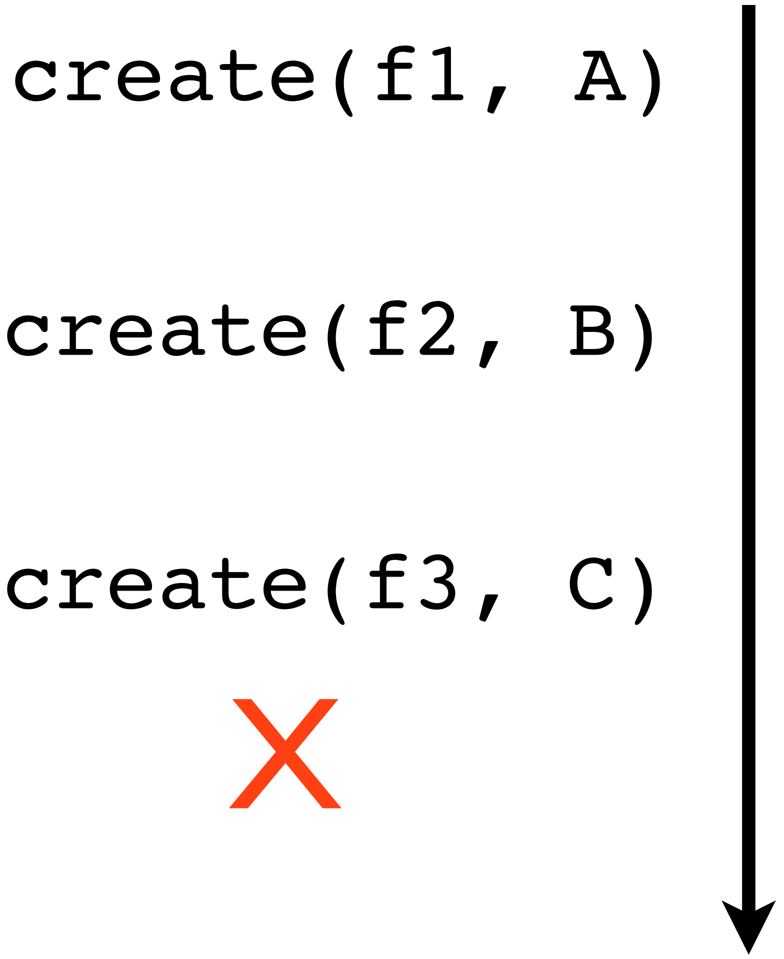


```
create(f1, A)

create(f2, B)

create(f3, C)
```

nosync, fsync and osync

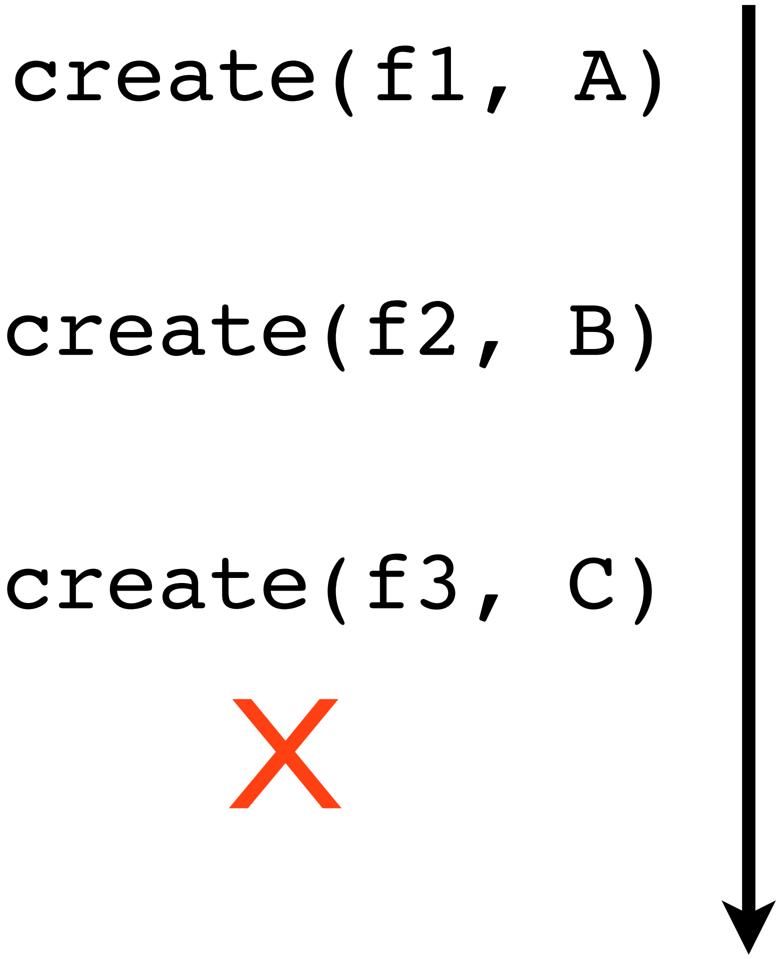


```
create(f1, A)

create(f2, B)

create(f3, C)
```

nosync, fsync and osync



```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

nosync, fsync and osync

```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

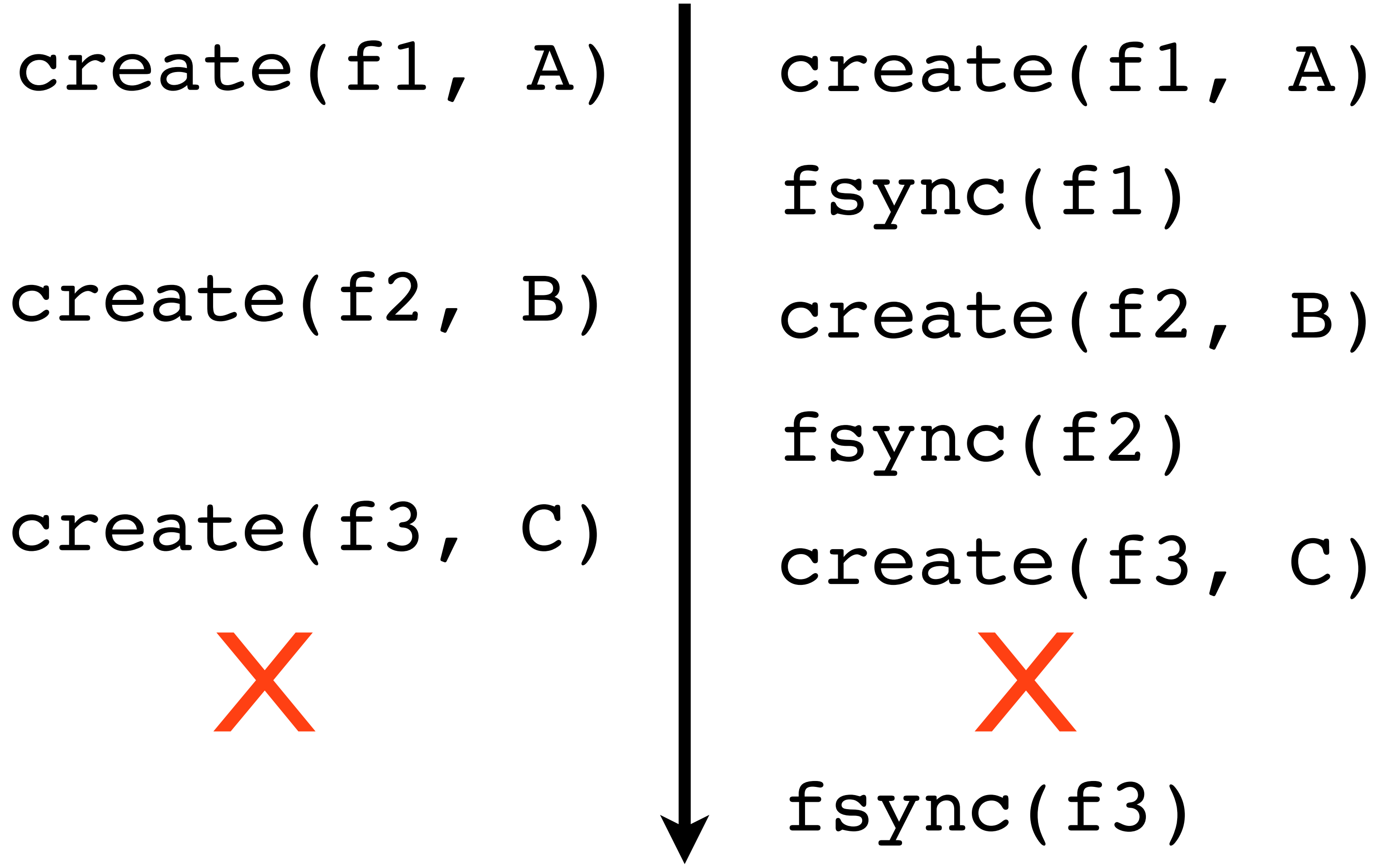
(A, ϕ, C)

(A, B, C)

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)

fsync(f3)
```

nosync, fsync and osync



```
create(f1, A)


create(f2, B)


create(f3, C)
```

X

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)
```

X

```
fsync(f3)
```

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

nosync, fsync and osync

```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)
```

X

```
fsync(f3)
```

Possible states

(A, B, ϕ)

(A, B, C)

nosync, fsync and osync

```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)
```

X

```
fsync(f3)
```

Possible states

(A, B, ϕ)

(A, B, C)

```
create(f1, A)
osync(f1)
create(f2, B)
osync(f2)
create(f3, C)


osync(f3)
```

nosync, fsync and osync

```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)
```

X

```
fsync(f3)
```

Possible states

(A, B, ϕ)

(A, B, C)

```
create(f1, A)
osync(f1)
create(f2, B)
osync(f2)
create(f3, C)
```

X

```
osync(f3)
```

nosync, fsync and osync

```
create(f1, A)

create(f2, B)

create(f3, C)
```

X

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)

(ϕ, ϕ, C) (ϕ, B, ϕ)

(ϕ, B, C) (A, B, ϕ)

(A, ϕ, C)

(A, B, C)

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
create(f3, C)
```

X

```
fsync(f3)
```

Possible states

(A, B, ϕ)

(A, B, C)

```
create(f1, A)
osync(f1)
create(f2, B)
osync(f2)
create(f3, C)
```

X

```
osync(f3)
```

Possible states

(ϕ, ϕ, ϕ)

(A, ϕ, ϕ)

(A, B, ϕ)

(A, B, C)

nosync, fsync and osync

```
create(f1, A)

create(f2, B)
```

```
create(f1, A)
fsync(f1)
create(f2, B)
fsync(f2)
```

```
create(f1, A)
osync(f1)
create(f2, B)
osync(f2)
C)
```

`osync()` ensures ordering and eventual durability

Possible states

(ϕ, ϕ, ϕ) (A, ϕ, ϕ)
(ϕ, ϕ, C) (ϕ, B, ϕ)
(ϕ, B, C) (A, B, ϕ)
(A, ϕ, C)
(A, B, C)

Possible states

(A, B, ϕ)
(A, B, C)

Possible states

(ϕ, ϕ, ϕ)
(A, ϕ, ϕ)
(A, B, ϕ)
(A, B, C)

osync() use cases

File formats like doc embed a number of files inside them [Harter11]

`osync()` use cases

File formats like `doc` embed a number of files inside them [Harter11]

```
write(body)
fsync(body)
write(header)
fsync(header)
```

`osync()` use cases

File formats like `doc` embed a number of files inside them [Harter11]

```
write(body)
fsync(body)
write(header)
fsync(header)
```

osync() use cases

File formats like doc embed a number of files inside them [Harter11]

```
write(body)
fsync(body)
write(header)
fsync(header)
```

➡

`osync()` use cases

File formats like `doc` embed a number of files inside them [Harter11]

```
write(body)
fsync(body)
write(header)
fsync(header)
```

➡

```
write(body)
osync(body)
write(header)
dsync(header)
```

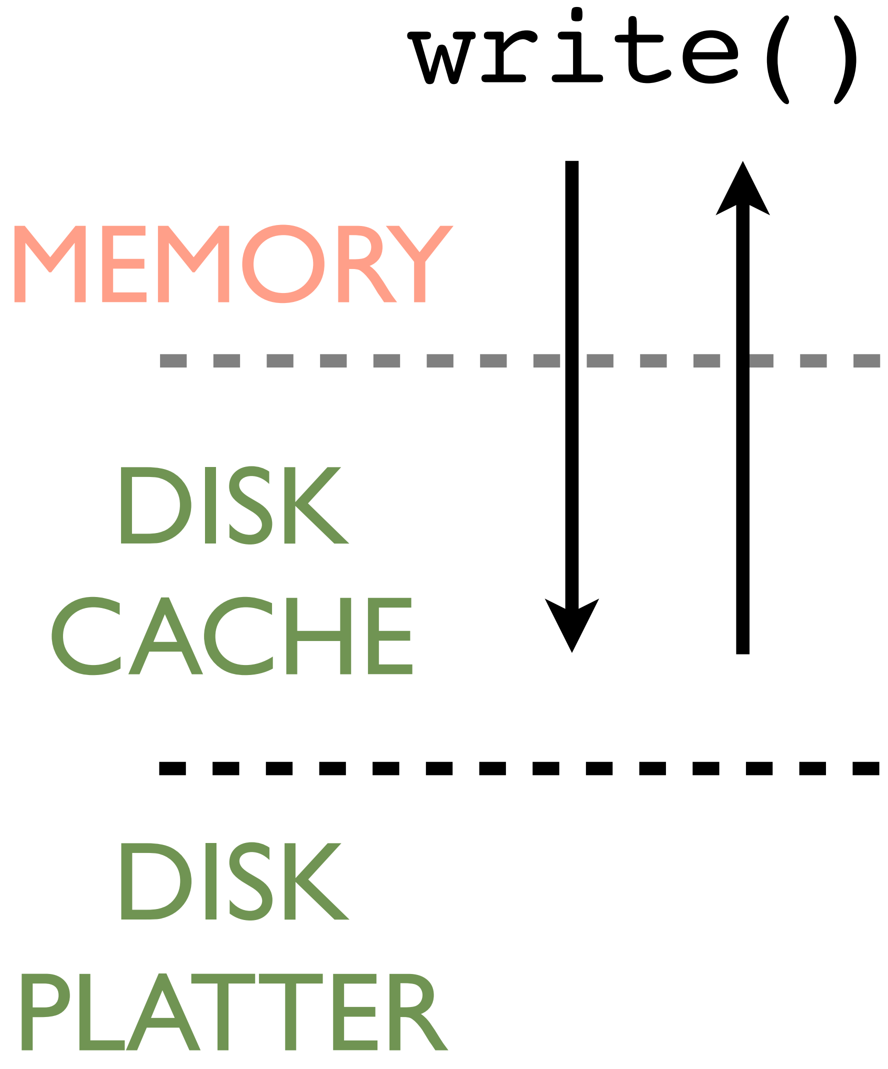
Asynchronous Durability Notifications

Asynchronous Durability Notifications

Conventional writes
return from the disk cache

Asynchronous Durability Notifications

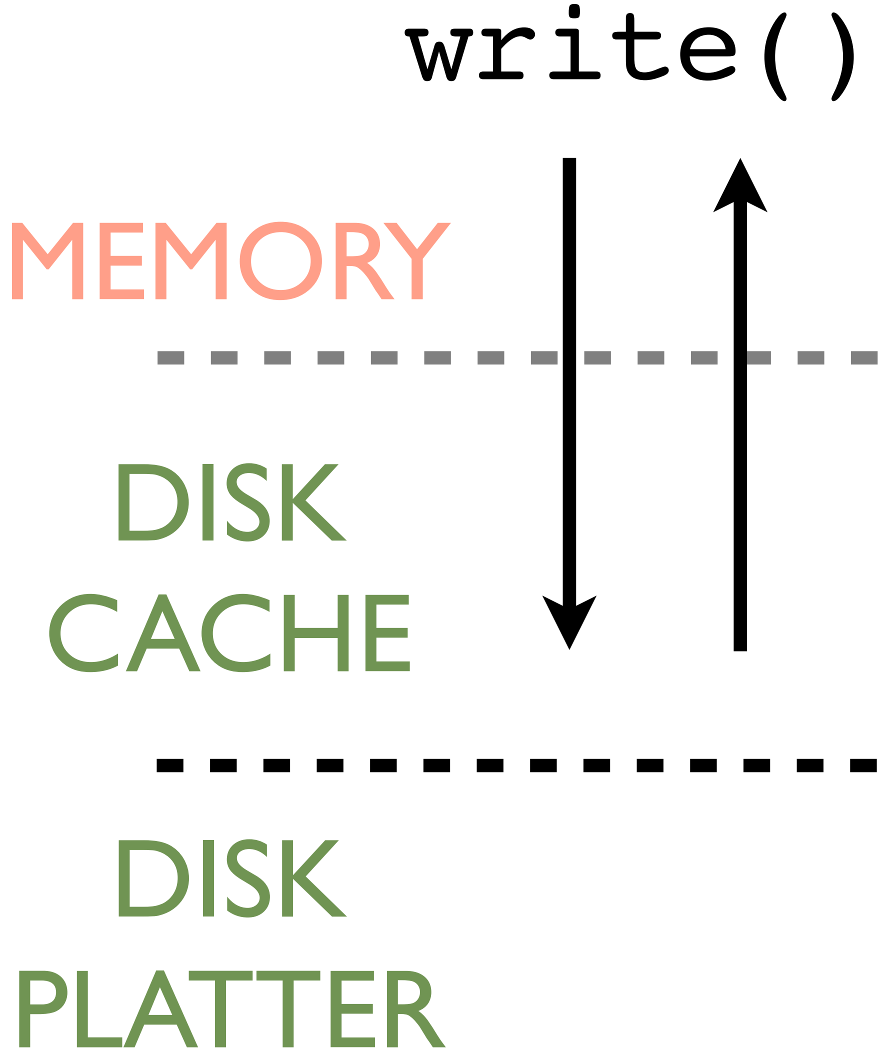
Conventional writes return from the disk cache



Asynchronous Durability Notifications

Conventional writes return from the disk cache

Flush command used to ensure durability

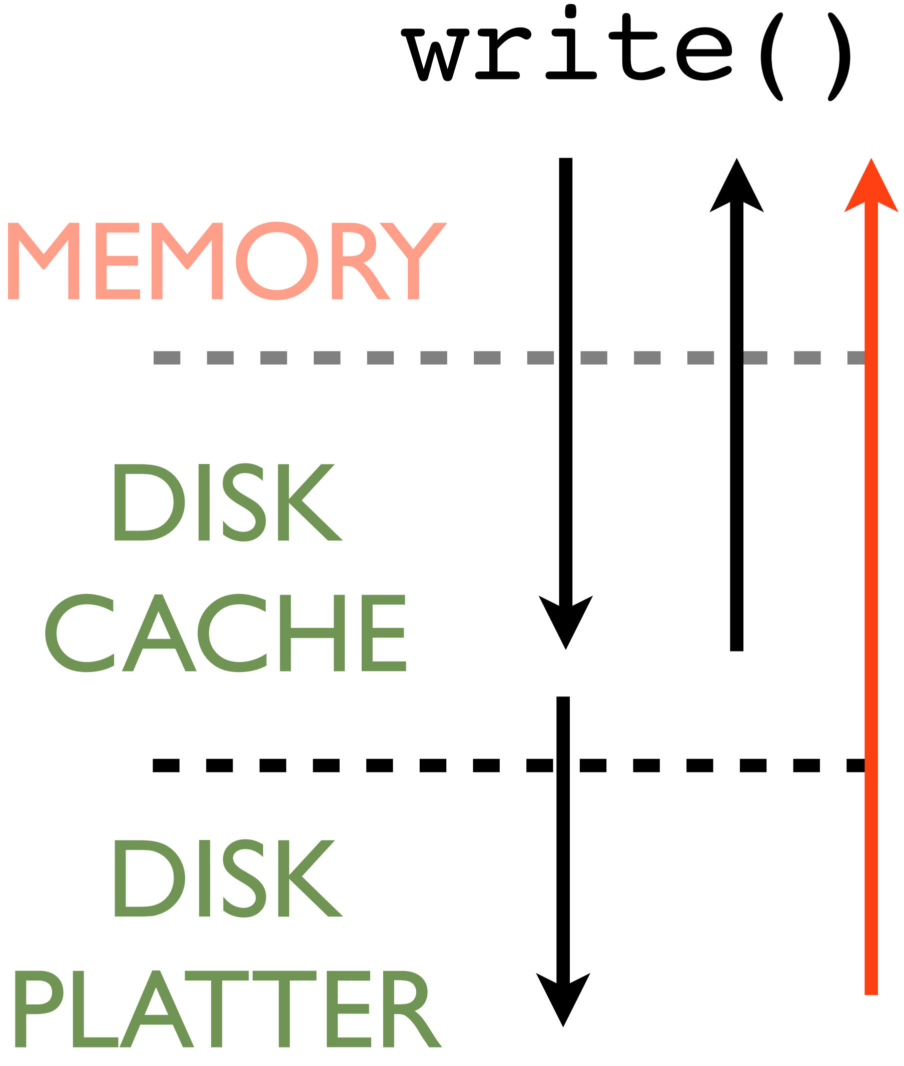


Asynchronous Durability Notifications

Conventional writes return from the disk cache

Flush command used to ensure durability

Asynchronous Durability notifications informs upper layer when blocks are durable



Optimistic Techniques

Optimistic Techniques

Early Commit	D Jc JM M Jc D JM M

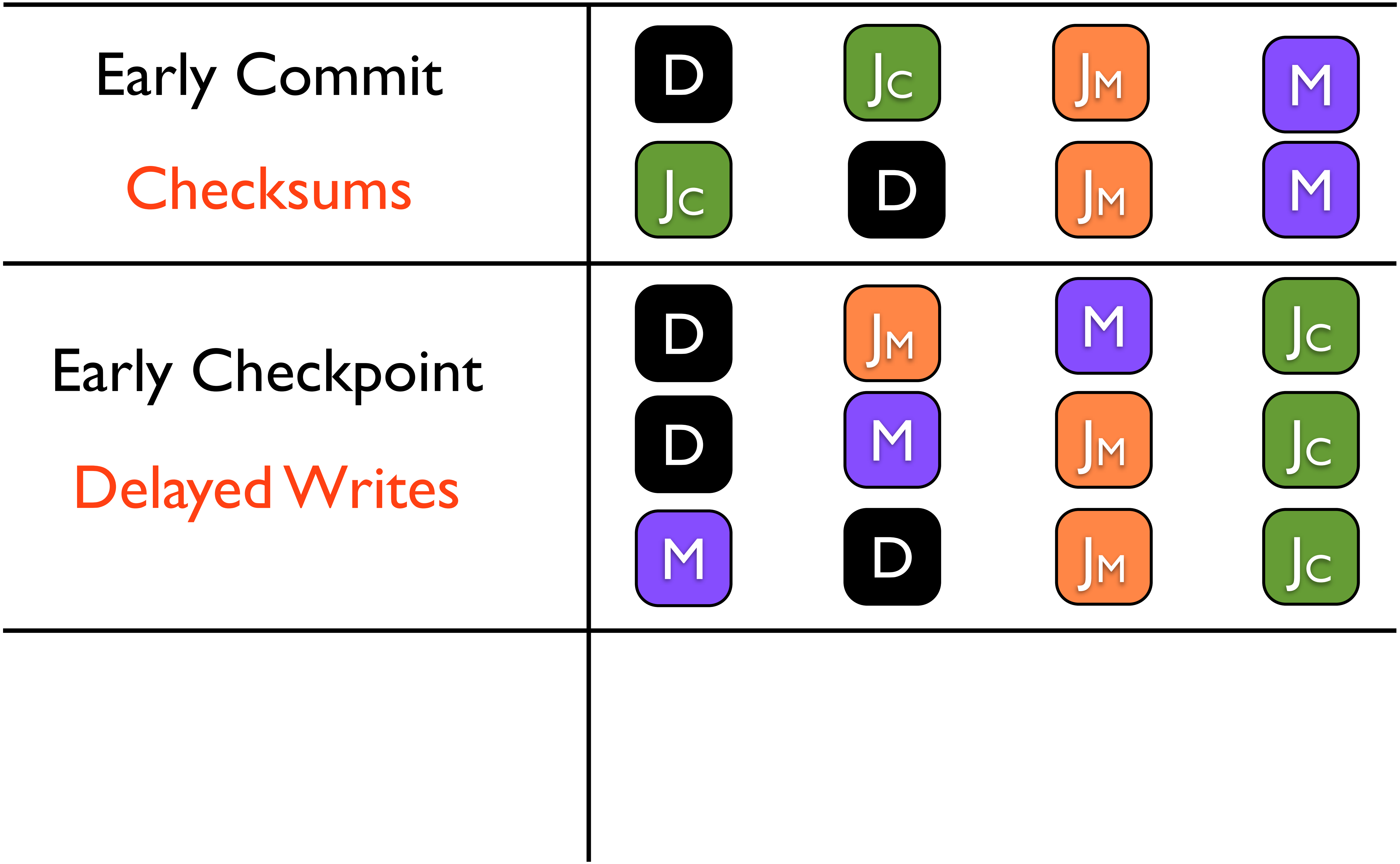
Optimistic Techniques

Early Commit	D	JC	JM	M
Checksums	JC	D	JM	M

Optimistic Techniques

Early Commit	D	JC	JM	M
Checksums	JC	D	JM	M
Early Checkpoint	D	JM	M	JC
	D	M	JM	JC
	M	D	JM	JC

Optimistic Techniques



Early Commit

Checksums

Early Checkpoint

Delayed Writes

Optimistic Techniques

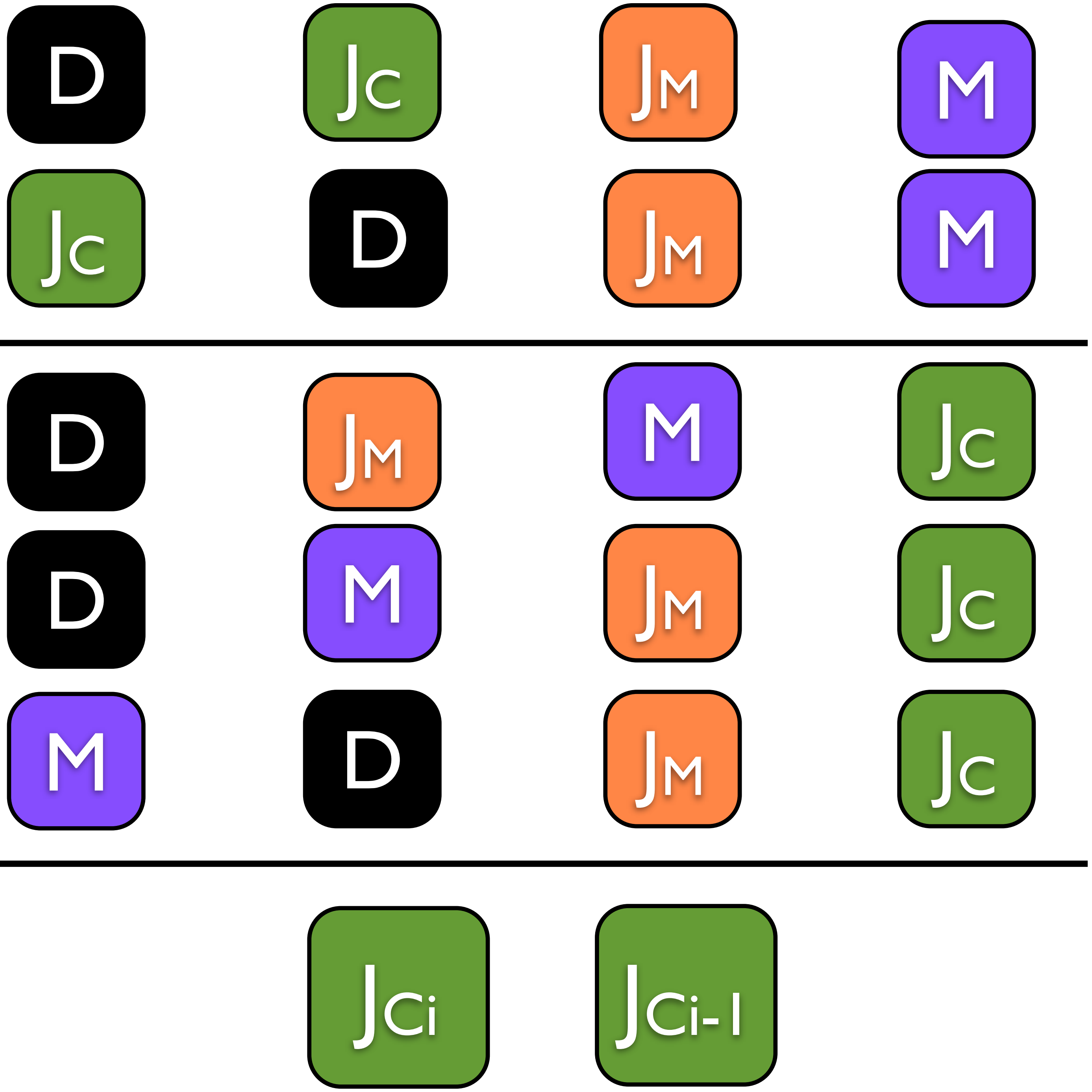
Early Commit

Checksums

Early Checkpoint

Delayed Writes

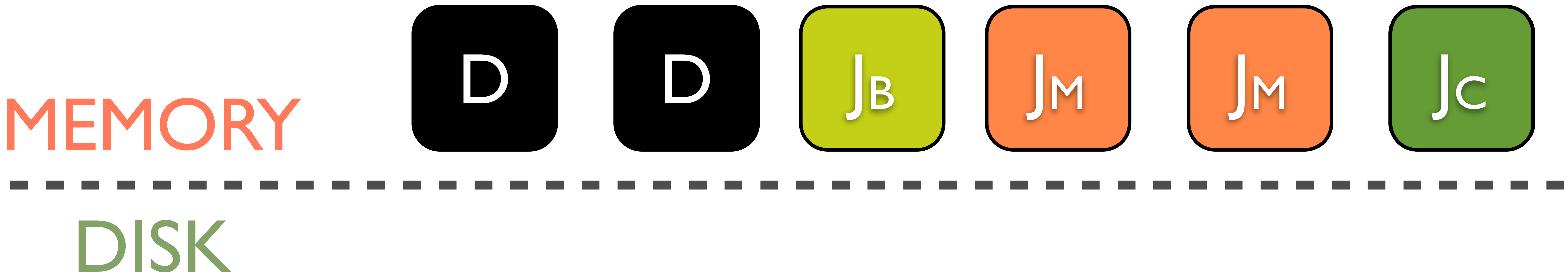
Transaction Misorder



Optimistic Techniques

Early Commit Checksums	D J_C J_M M J_C D J_M M
Early Checkpoint Delayed Writes	D J_M M J_C D M J_M J_C M D J_M J_C
Transaction Misorder In-order Journal Replay & Recovery	J_{C_i} $J_{C_{i-1}}$

Allowing re-ordering



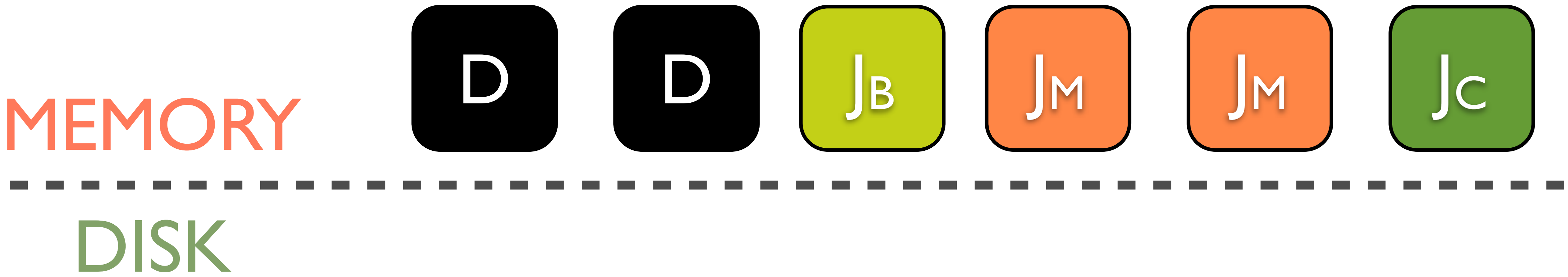
MEMORY

D D J_B J_M J_M J_C

DISK

Allowing re-ordering

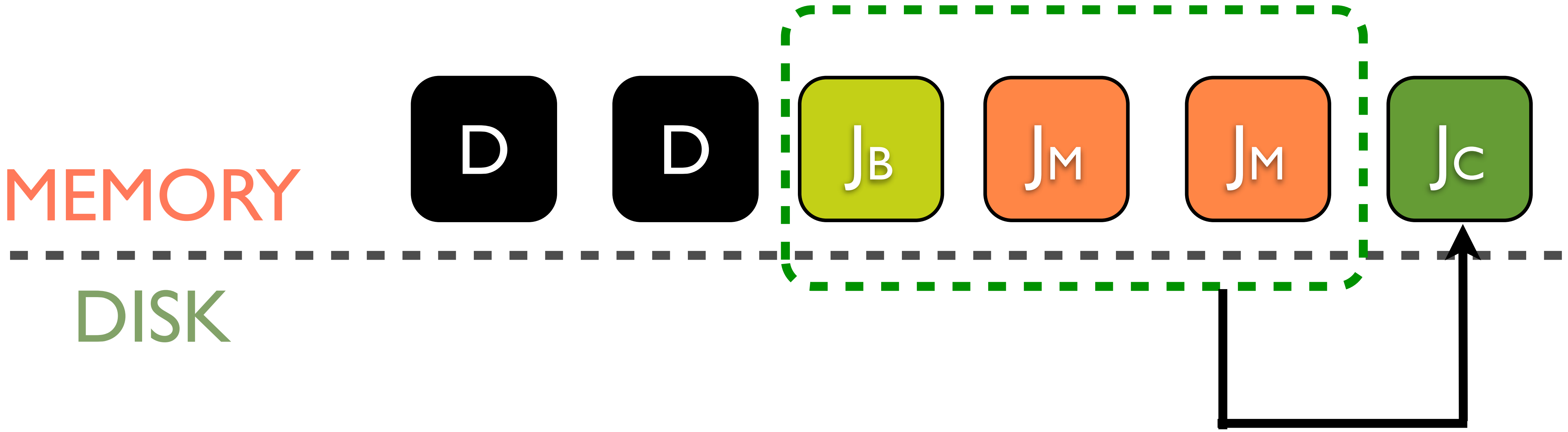
We use two checksums to detect mis-ordering upon crash



Allowing re-ordering

We use two checksums to detect mis-ordering upon crash

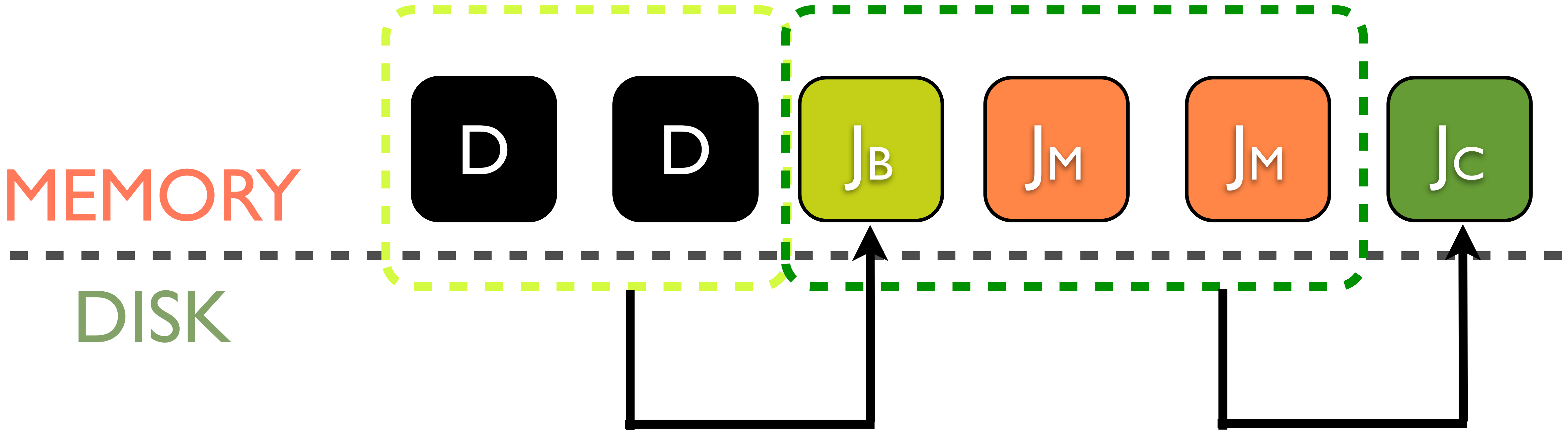
- Metadata transactional checksum [Prabhakaran05]



Allowing re-ordering

We use two checksums to detect mis-ordering upon crash

- Metadata transactional checksum [Prabhakaran05]
- Data transactional checksum



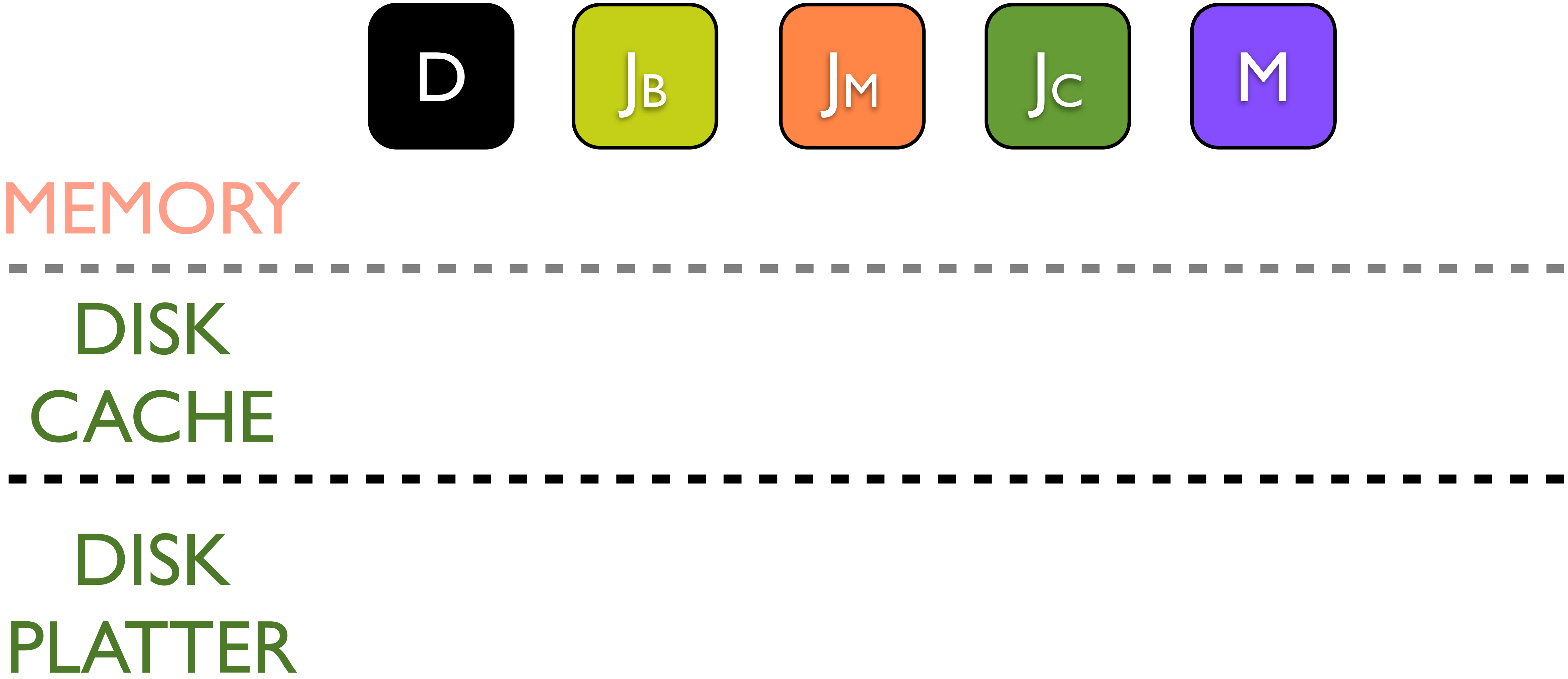
Avoiding re-ordering

We use durability notifications to know when writes leave the disk cache

We avoid having writes we don't want re-ordered in the disk cache at the same time

Avoiding re-ordering

Example: checkpoint writes



D

J_B

J_M

J_C

M

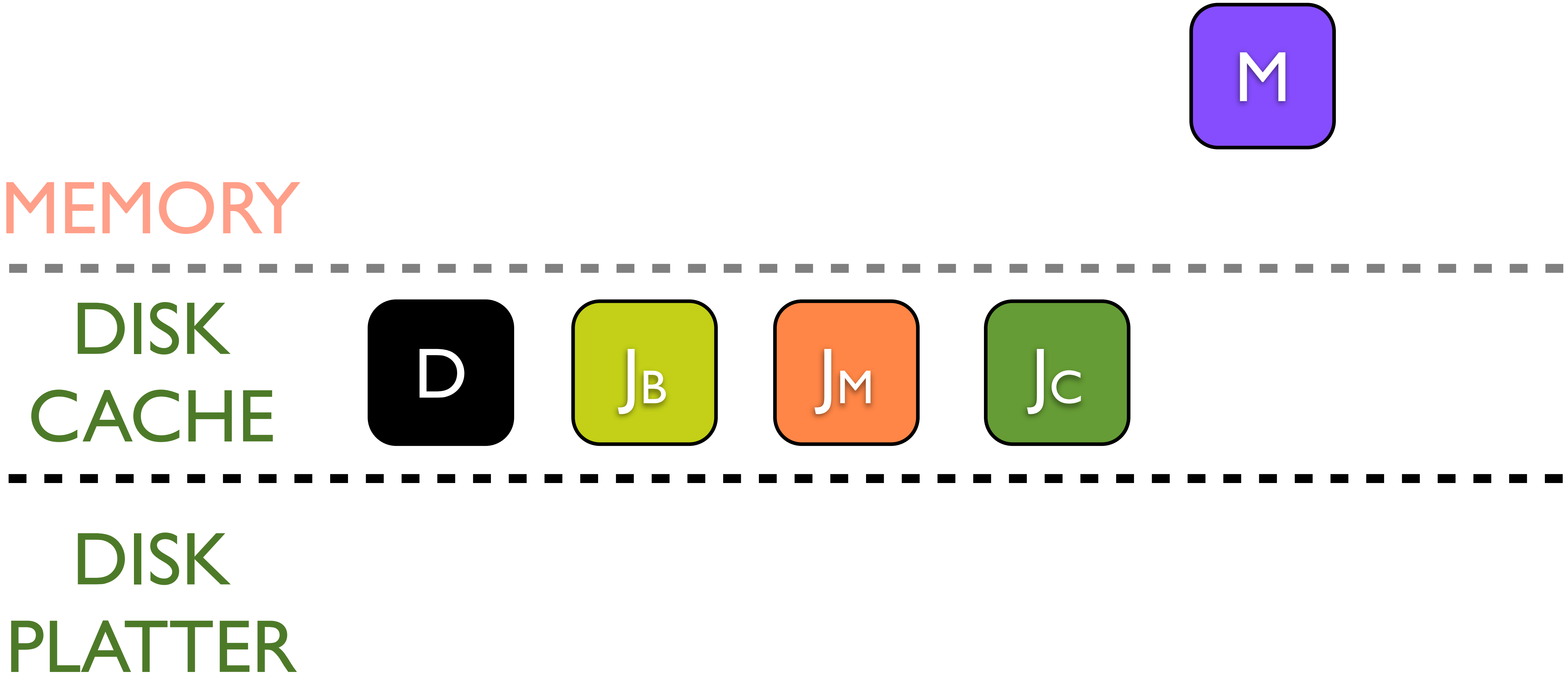
MEMORY

DISK CACHE

DISK PLATTER

Avoiding re-ordering

Example: checkpoint writes



M

MEMORY

DISK CACHE

D

JB

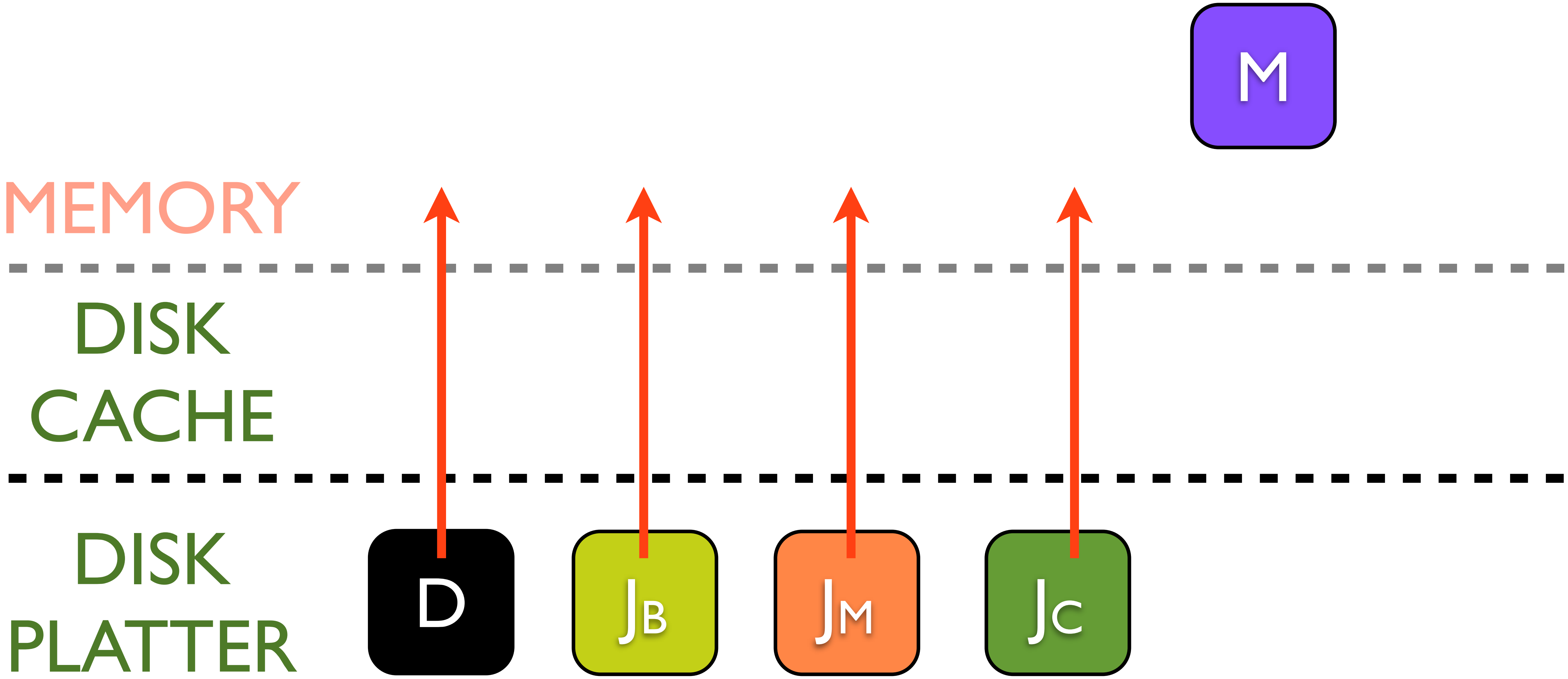
JM

JC

DISK PLATTER

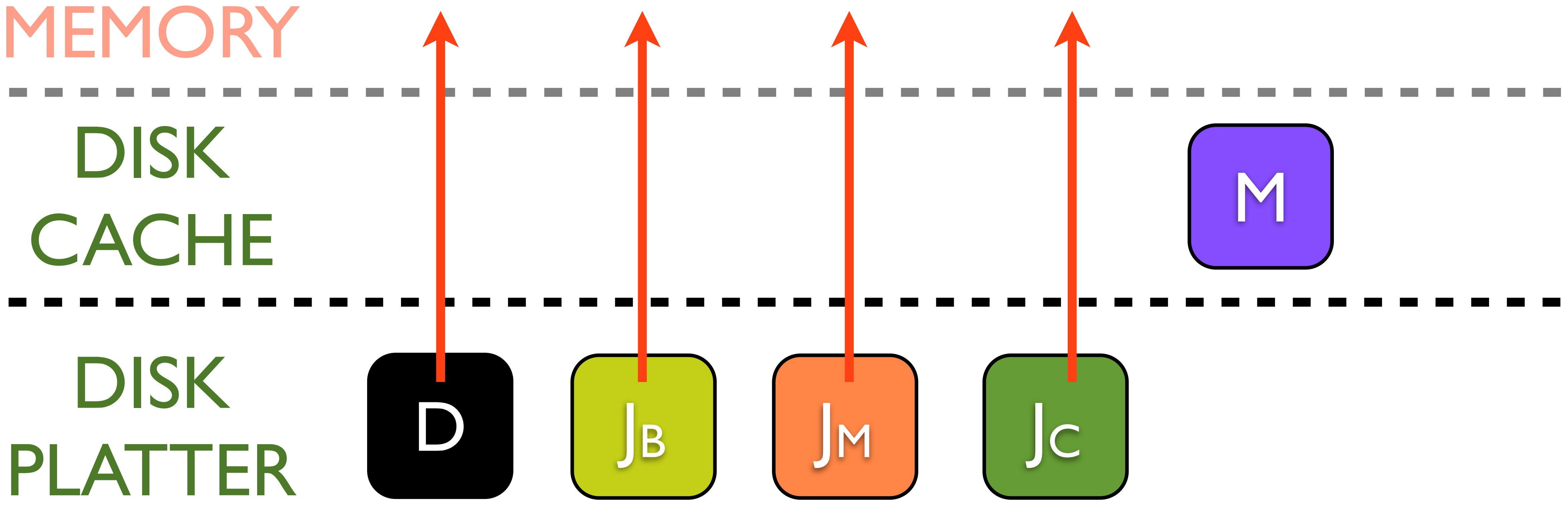
Avoiding re-ordering

Example: checkpoint writes



Avoiding re-ordering

Example: checkpoint writes



Avoiding re-ordering

Avoiding re-ordering

We delay checkpoint writes:

- Write checkpoint blocks only after the entire transaction is durable
- Checkpoint transactions in order

Avoiding re-ordering

We delay checkpoint writes:

- Write checkpoint blocks only after the entire transaction is durable
- Checkpoint transactions in order

We delay freeing journal blocks:

- Free journal blocks only after entire transaction has been durably checkpointed
- Free journal transaction blocks in order

In-order recovery

In-order recovery

After a crash, we recover journal transactions in order

In-order recovery

After a crash, we recover journal transactions in order

Eligible transactions are replayed

In-order recovery

After a crash, we recover journal transactions in order

Eligible transactions are replayed

Recovery stops at first corrupt transaction

In-order recovery

After a crash, we recover journal transactions in order

Eligible transactions are replayed

Recovery stops at first corrupt transaction

In order to be replayed, transaction has to match all checksums

In-order recovery

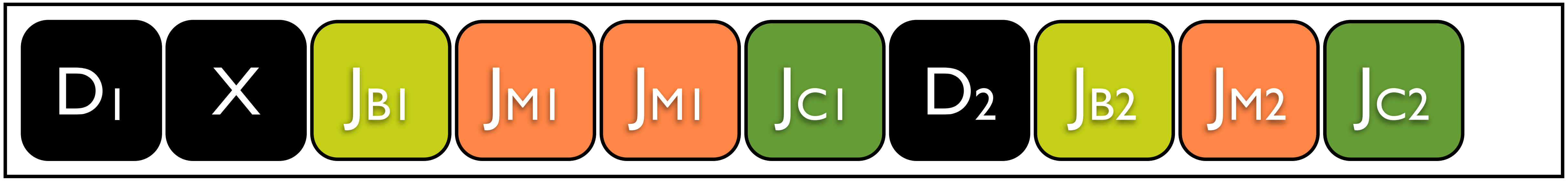
After a crash, we recover journal transactions in order

Eligible transactions are replayed

Recovery stops at first corrupt transaction

In order to be replayed, transaction has to match all checksums

On-disk journal



D_1 X J_{B1} J_{M1} J_{M1} J_{C1} D_2 J_{B2} J_{M2} J_{C2}

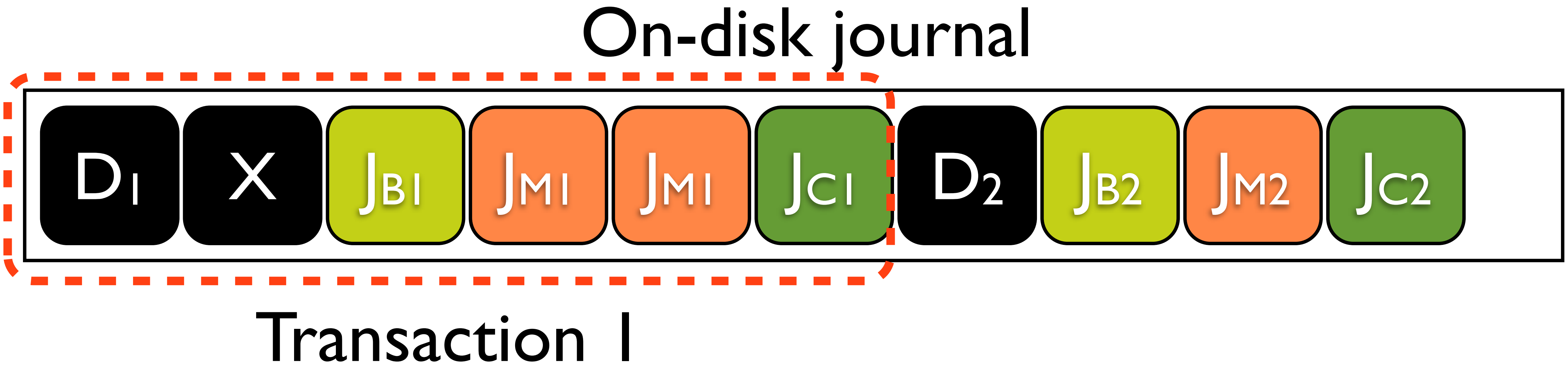
In-order recovery

After a crash, we recover journal transactions in order

Eligible transactions are replayed

Recovery stops at first corrupt transaction

In order to be replayed, transaction has to match all checksums



On-disk journal

Transaction 1

In-order recovery

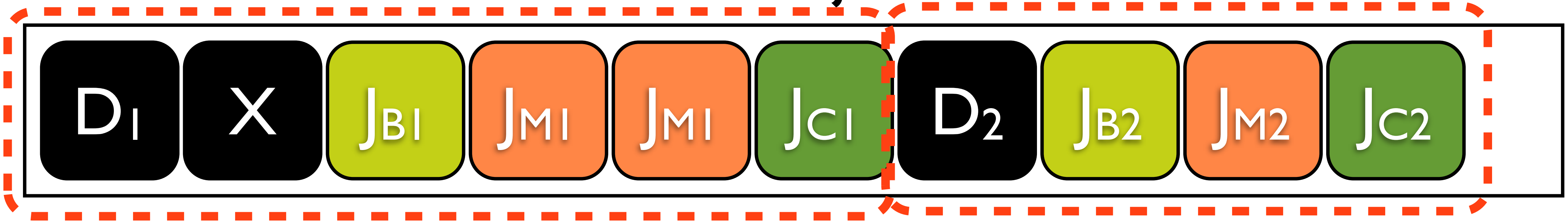
After a crash, we recover journal transactions in order

Eligible transactions are replayed

Recovery stops at first corrupt transaction

In order to be replayed, transaction has to match all checksums

On-disk journal



Transaction 1

Transaction 2

Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back

D JM JC M

Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back

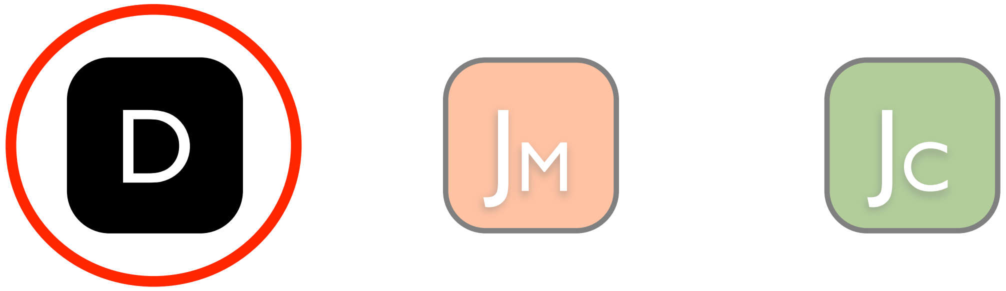
Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back

D JM JC

Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back



Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back

Handling Data Overwrites

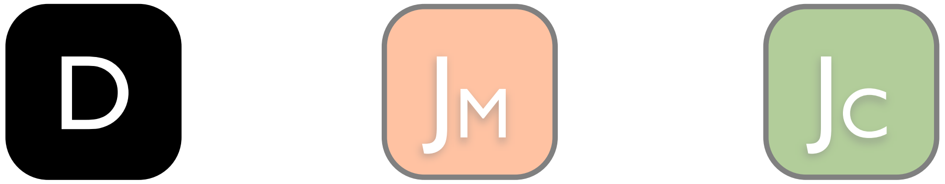
In ordered journaling mode, even if tx fails, data block cannot be rolled back

D JM JC

This is not a problem for new data blocks

Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back



This is not a problem for new data blocks

For overwritten data blocks, old metadata still point to them

Handling Data Overwrites

In ordered journaling mode, even if tx fails, data block cannot be rolled back

This is not a problem for new data blocks

For overwritten data blocks, old metadata still point to them

We handle this by journaling only overwritten data blocks

Handling Data Overwrites

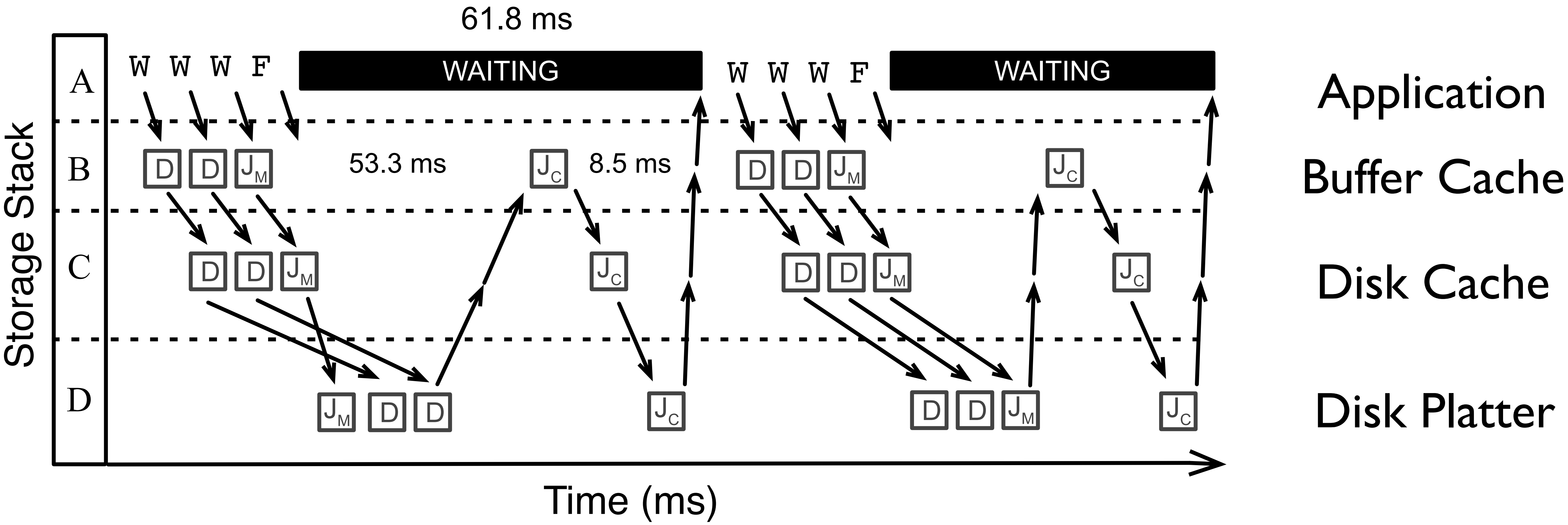
In ordered journaling mode, even if tx fails, data block cannot be rolled back

This is not a problem for new data blocks

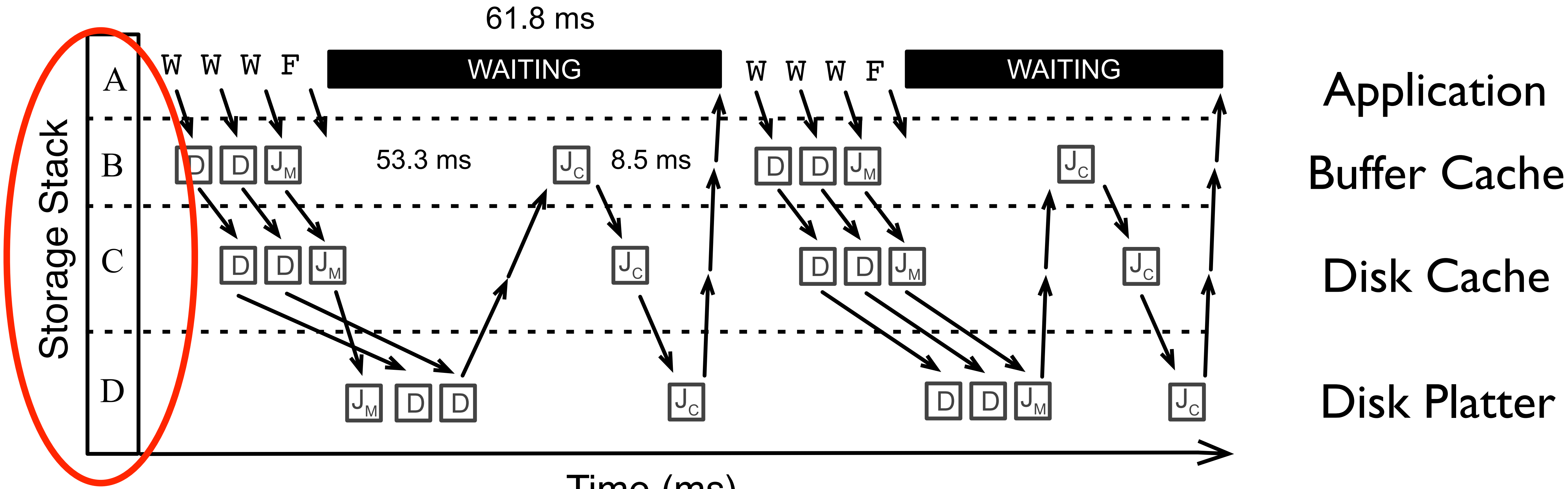
For overwritten data blocks, old metadata still point to them

We handle this by journaling only overwritten data blocks

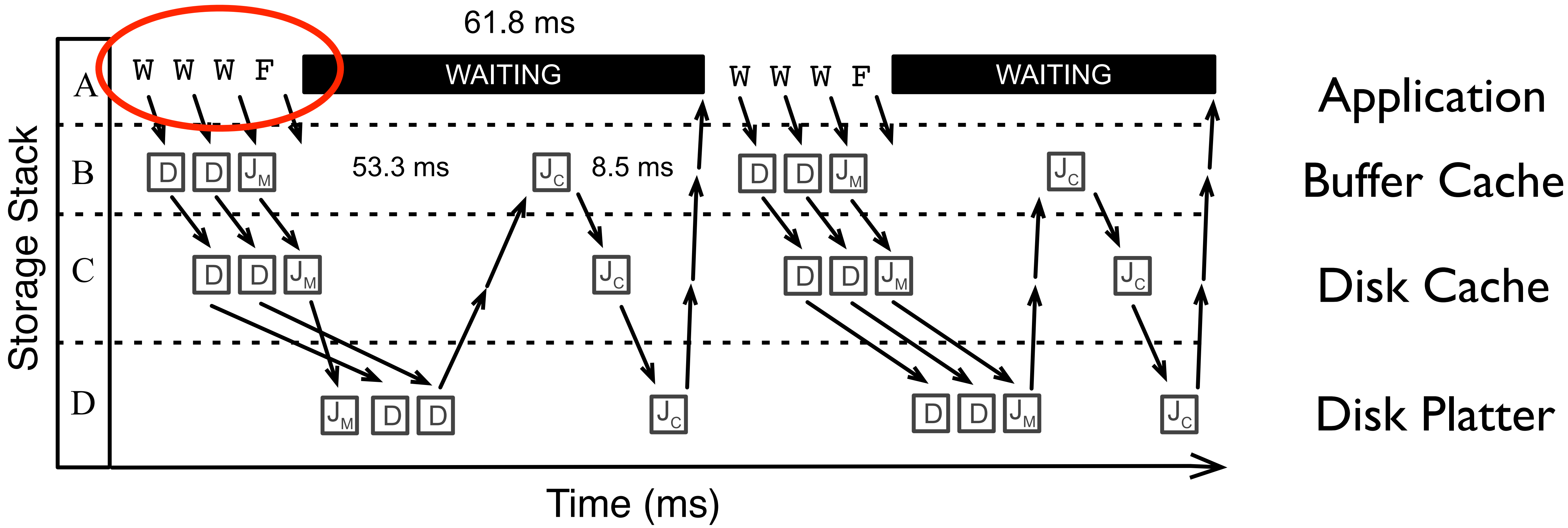
Using checksums, delayed writes and in-order recovery, the optimistic protocol ensures consistency without flushing



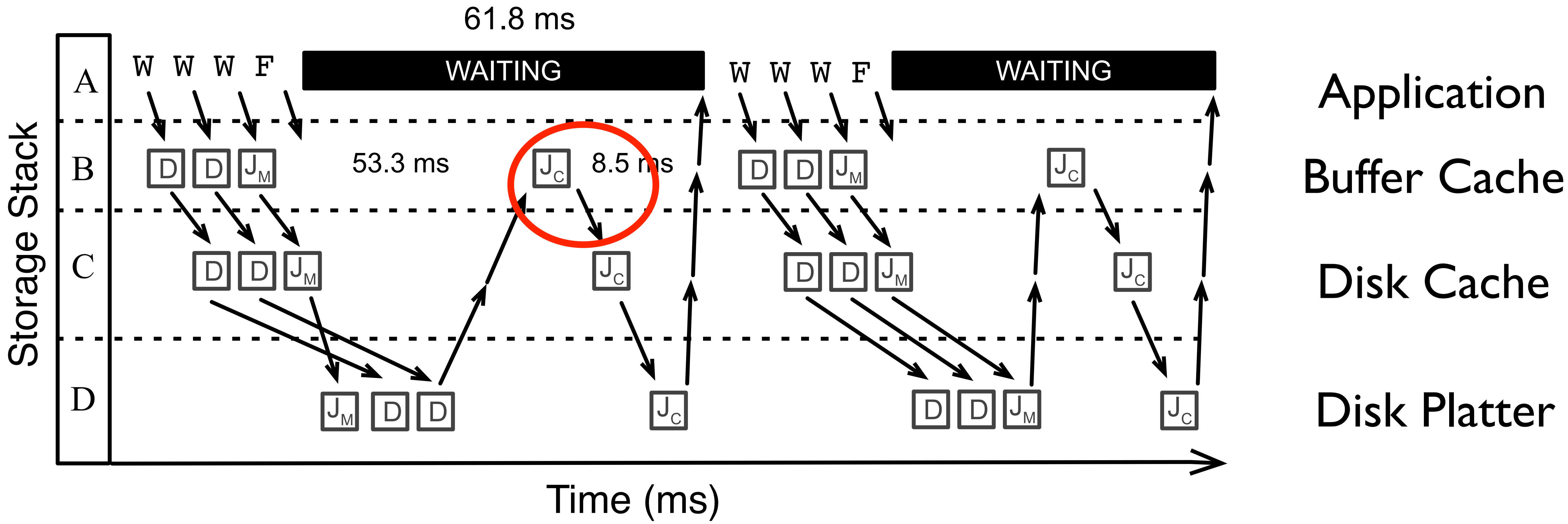
ext4 with barriers



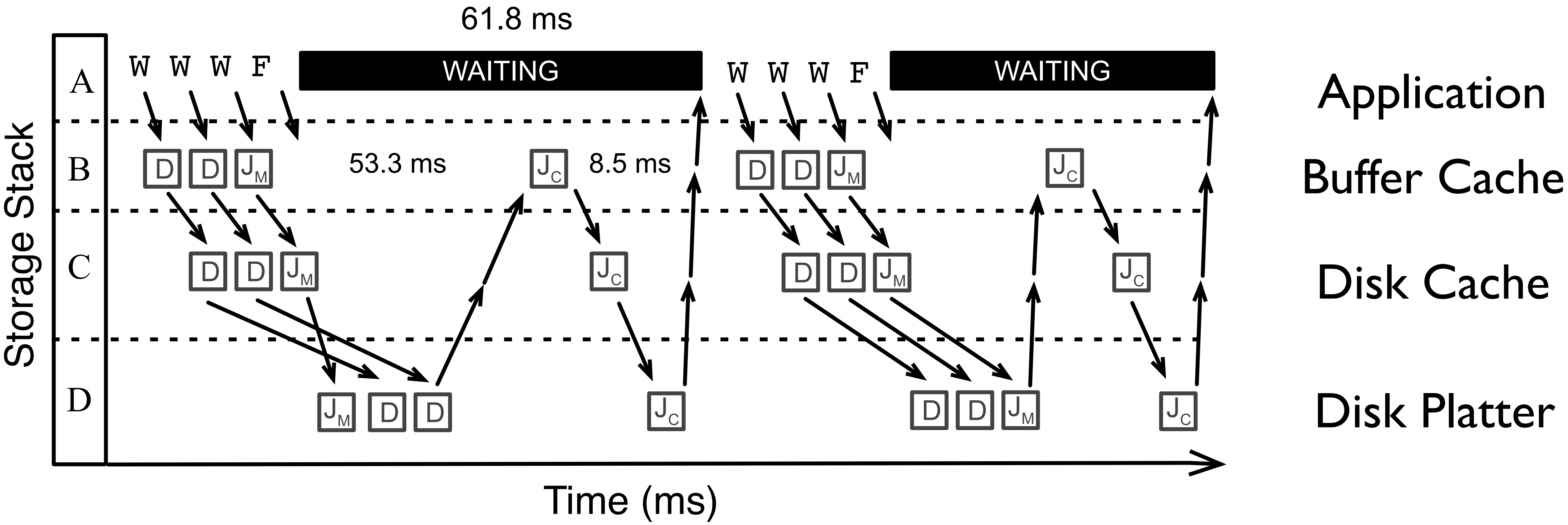
ext4 with barriers



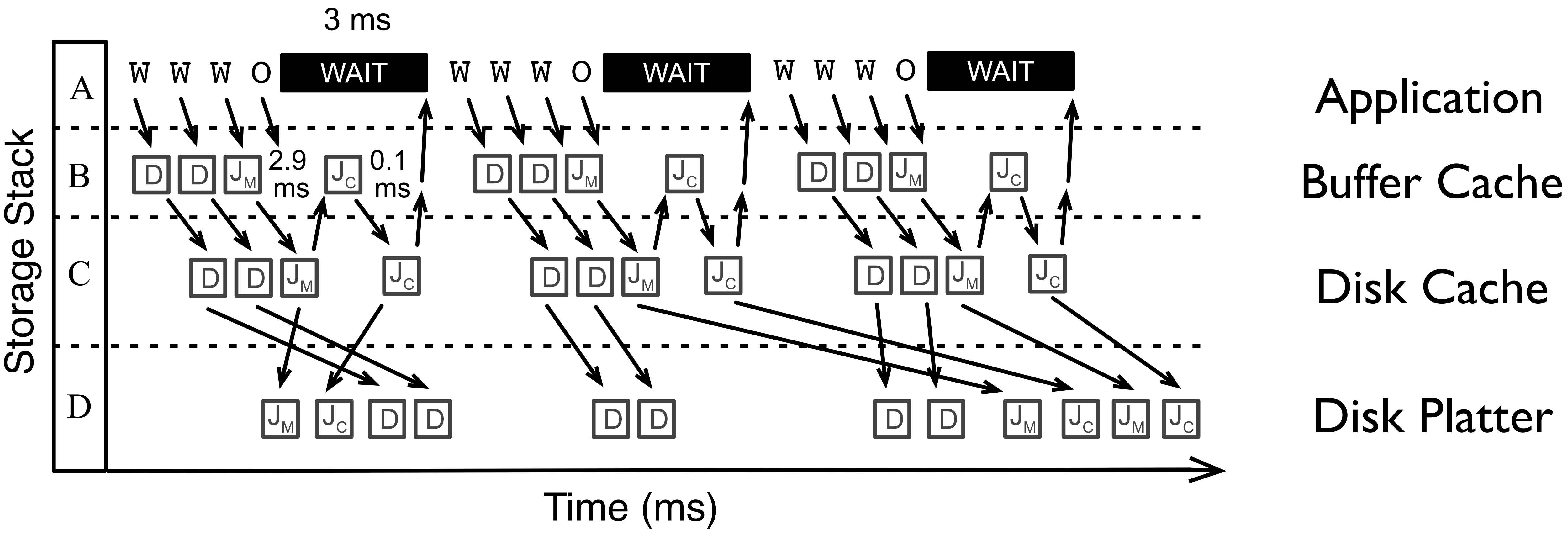
ext4 with barriers



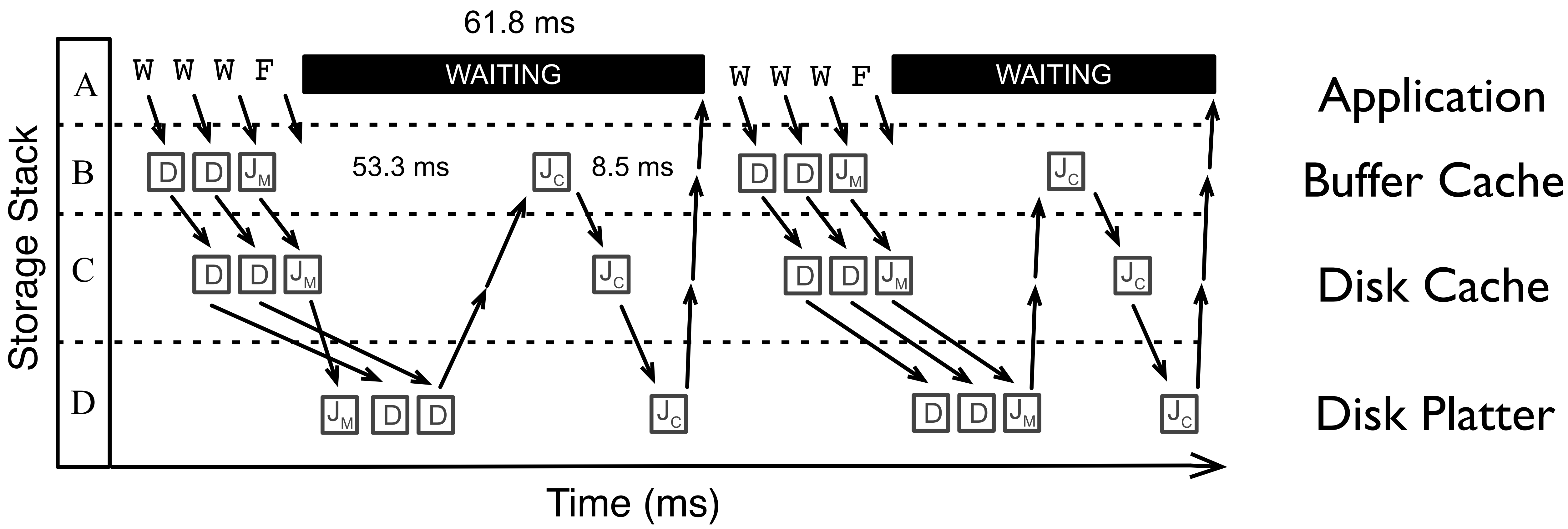
ext4 with barriers



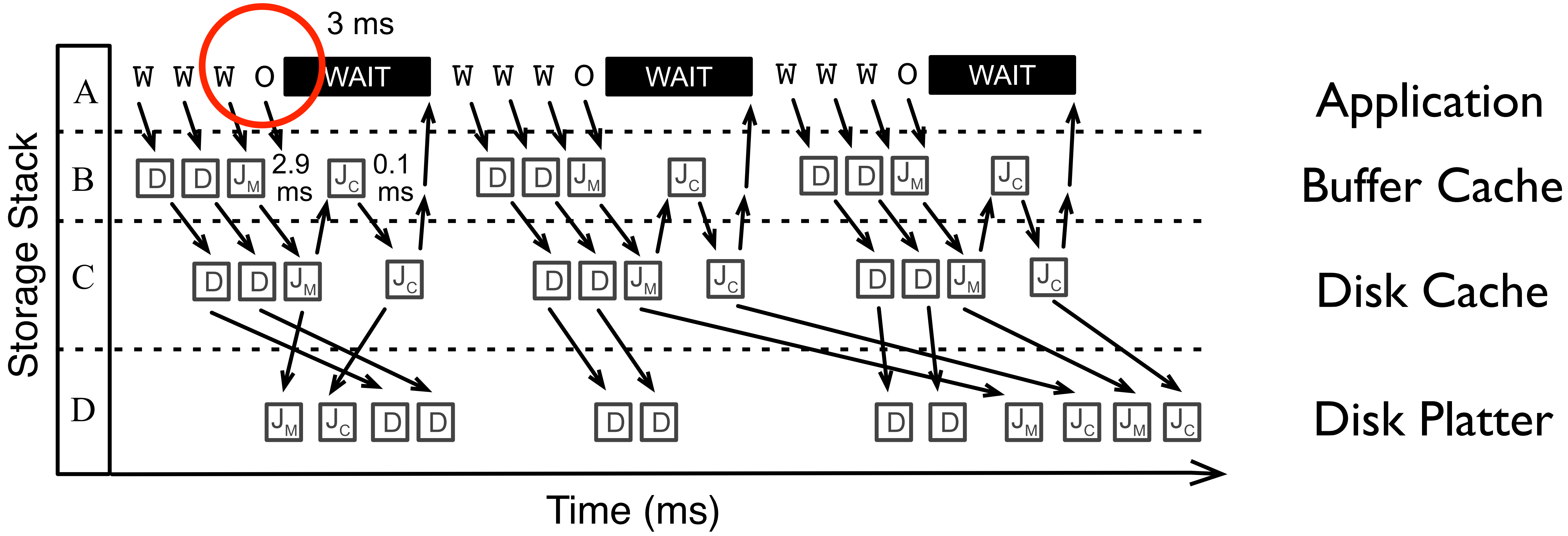
ext4 with barriers



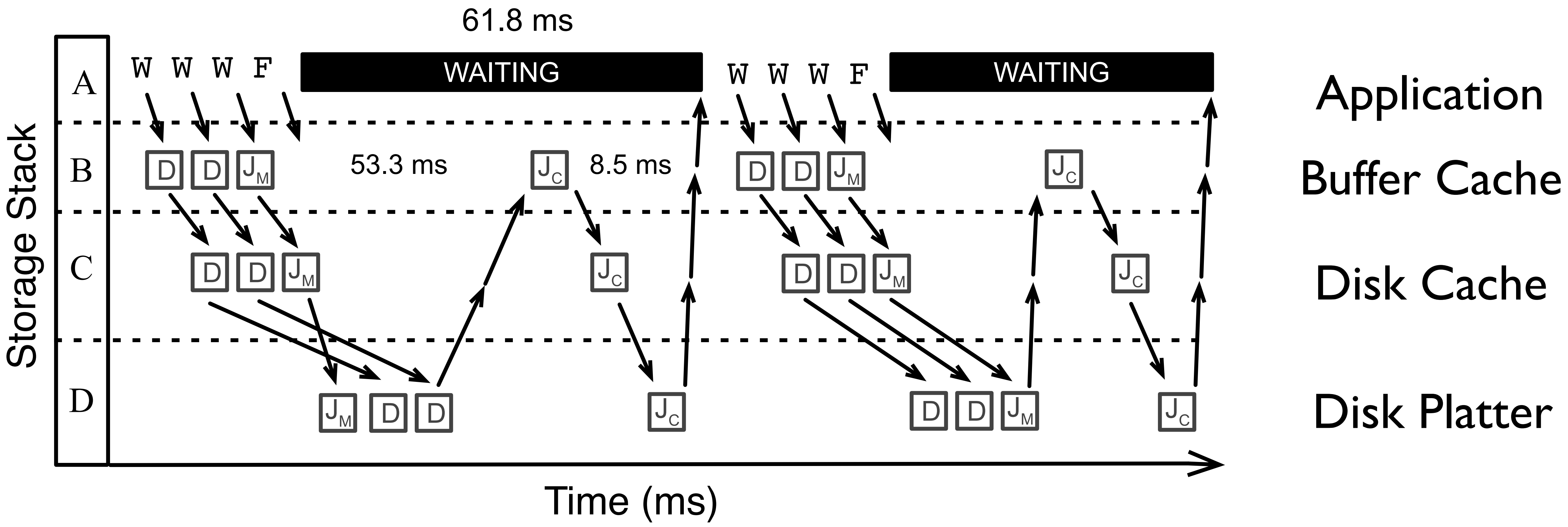
OptFS



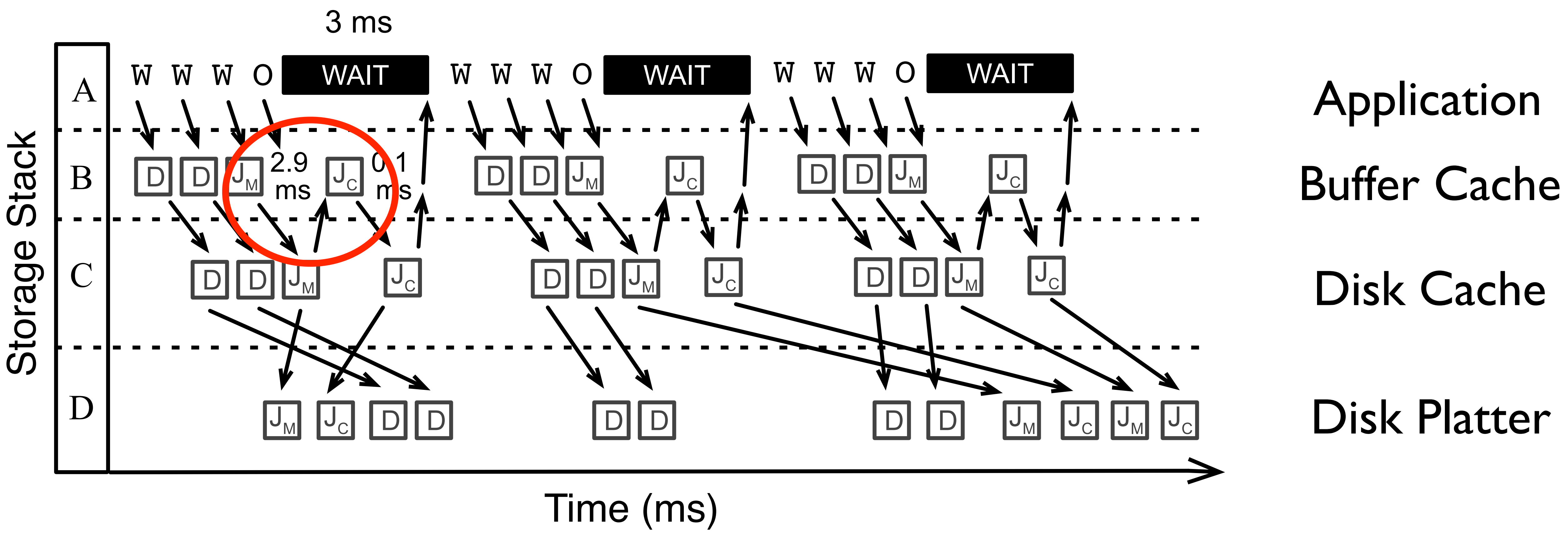
ext4 with barriers



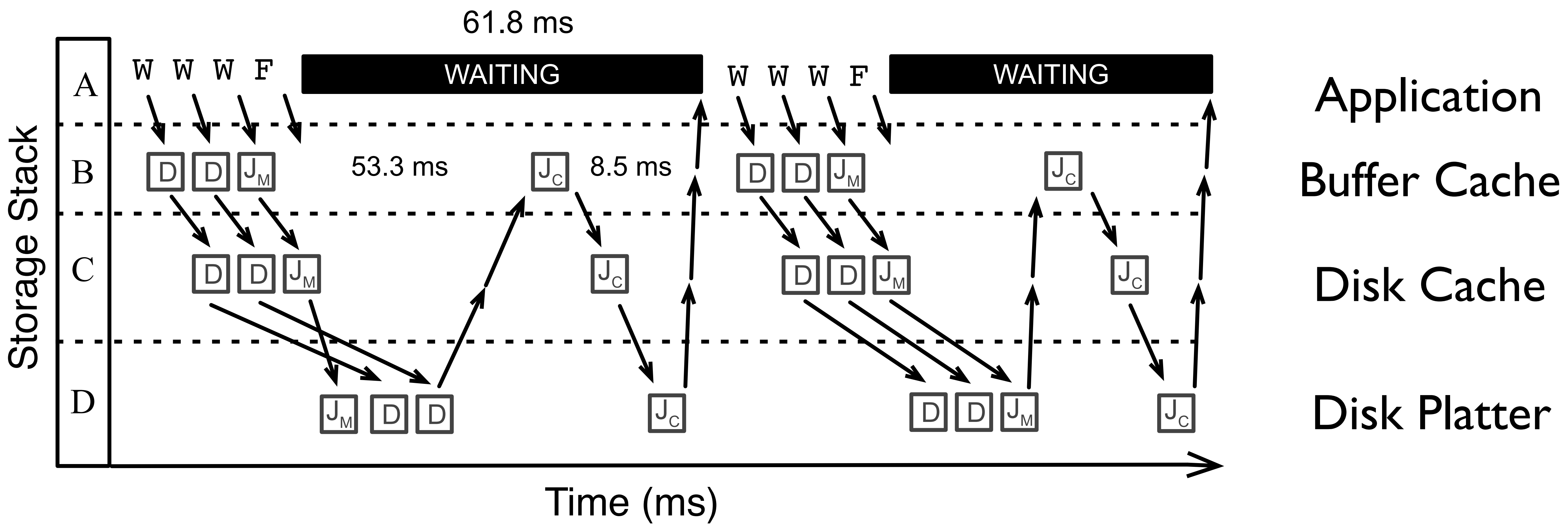
OptFS



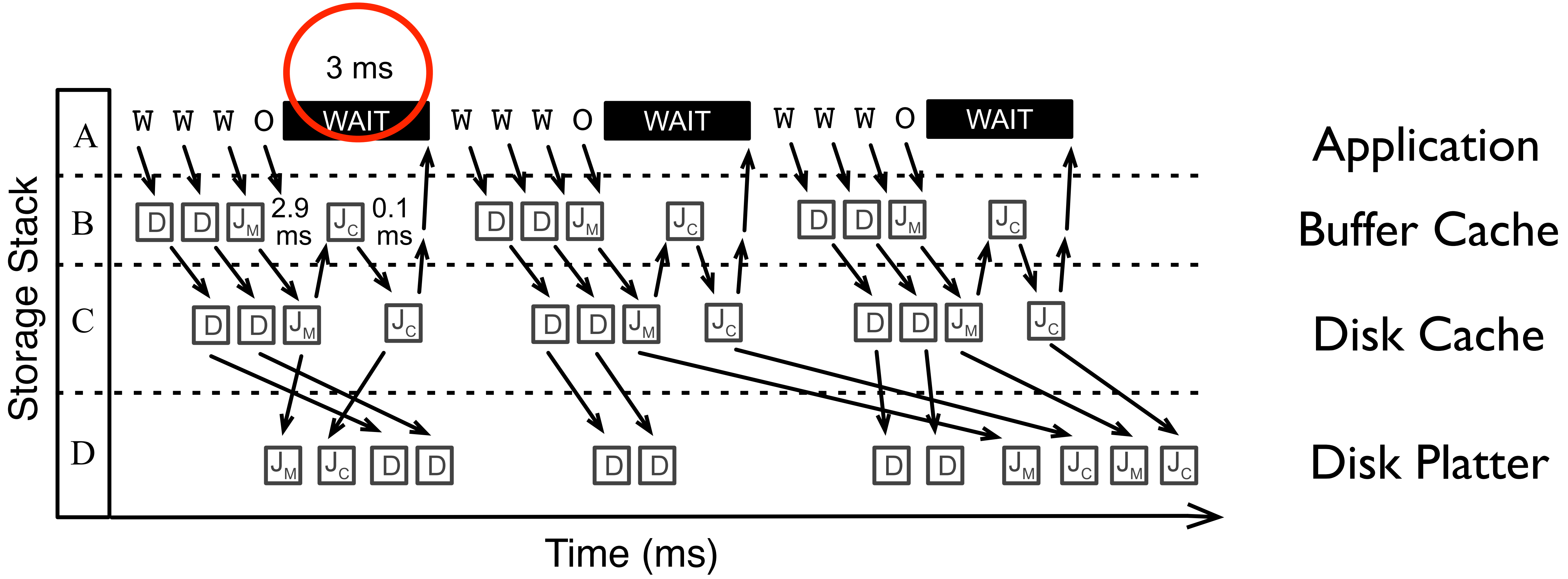
ext4 with barriers



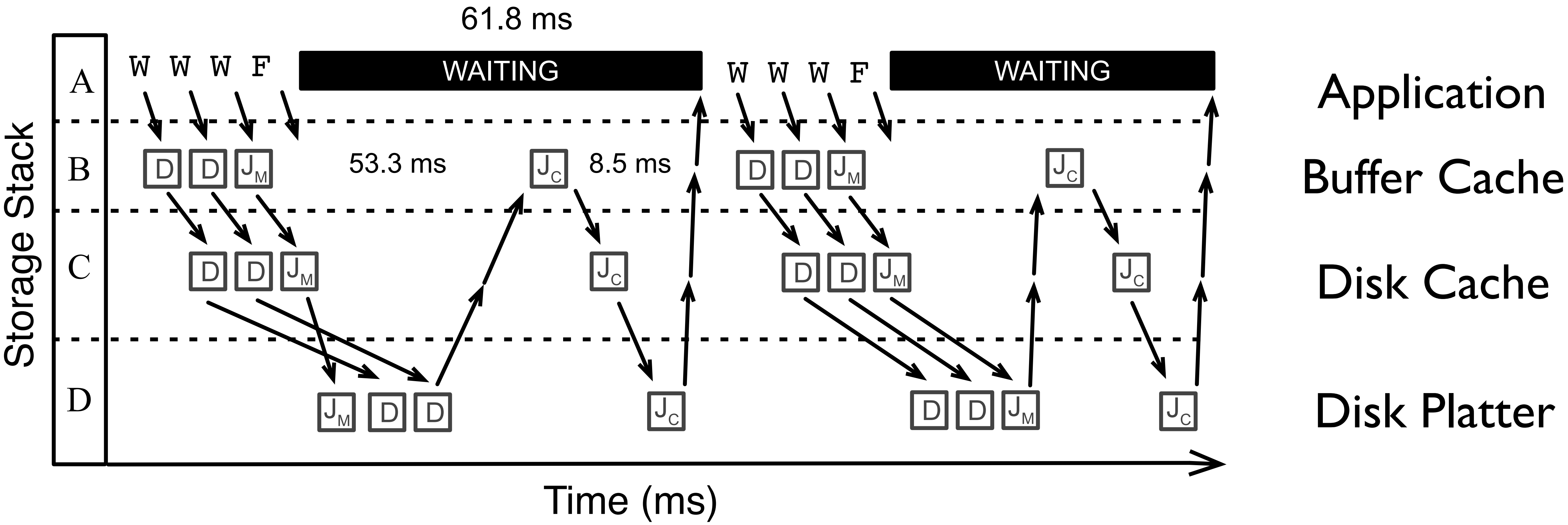
OptFS



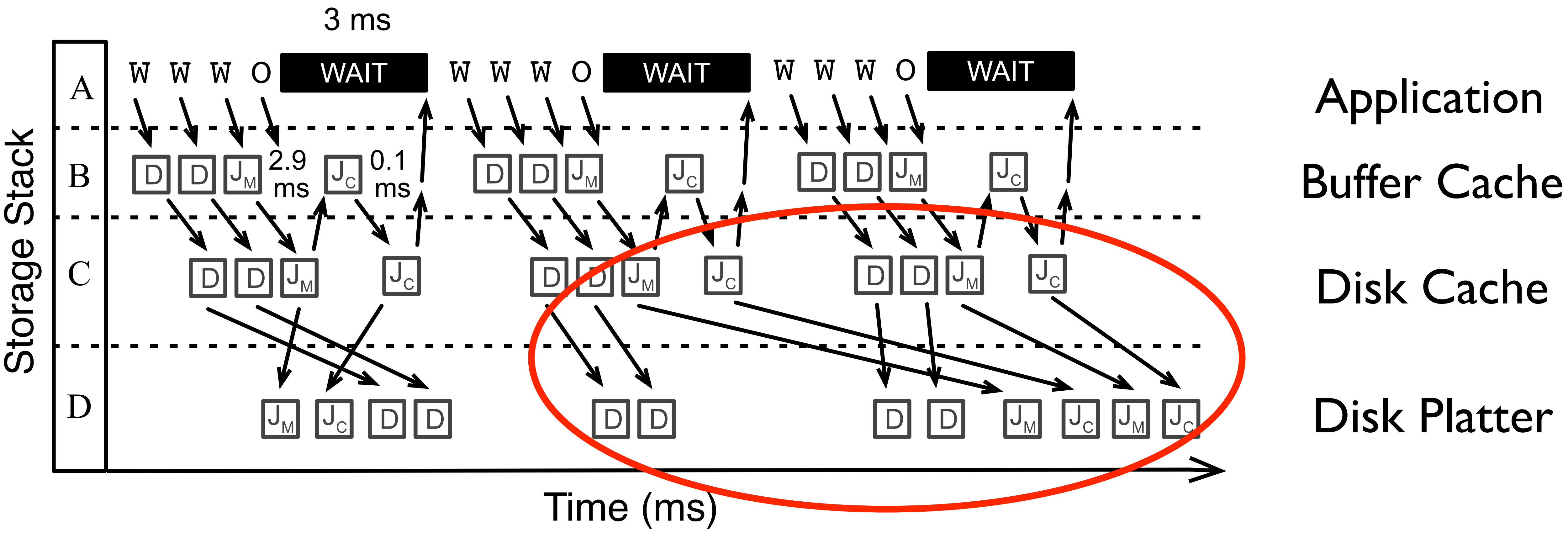
ext4 with barriers



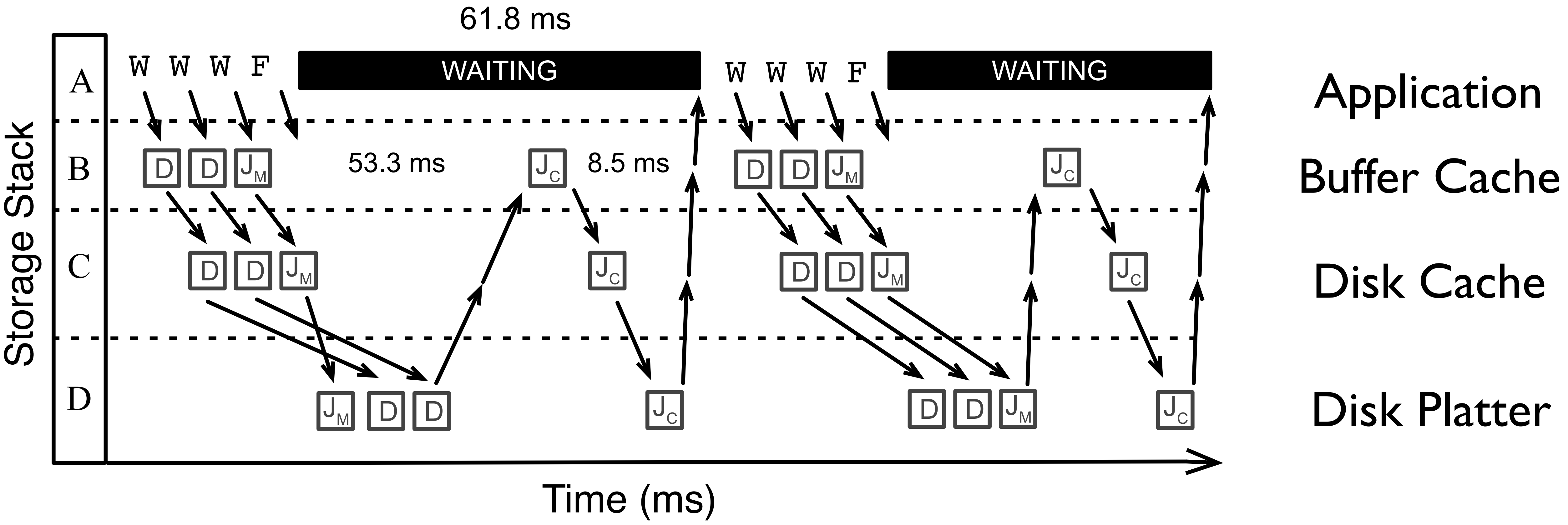
OptFS



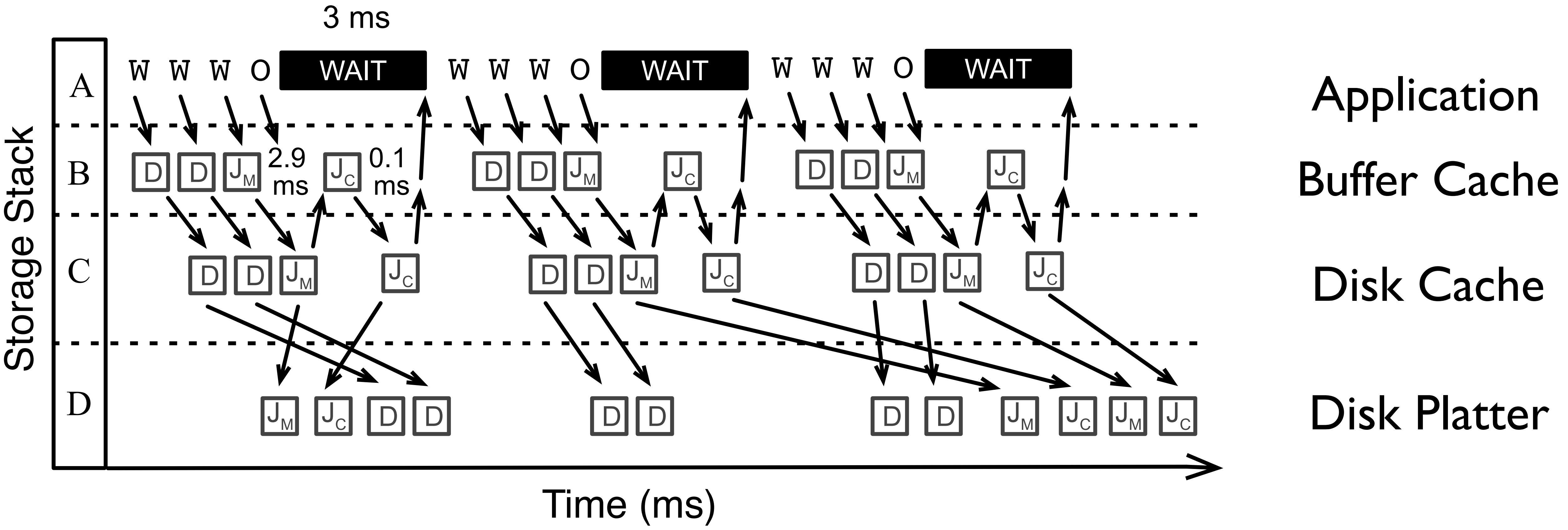
ext4 with barriers



OptFS



ext4 with barriers



OptFS

Implementation

Implementation

We implemented the Optimistic File System (OptFS)

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Linux kernel: 3.2

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Linux kernel: 3.2

Lines added/modified: 2400

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Linux kernel: 3.2

Lines added/modified: 2400

Source code available:

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Linux kernel: 3.2

Lines added/modified: 2400

Source code available:

- http://research.cs.wisc.edu/adsl/Software/optfs/

Implementation

We implemented the Optimistic File System (OptFS)

OptFS is based on ext4 code

Linux kernel: 3.2

Lines added/modified: 2400

Source code available:

- http://research.cs.wisc.edu/adsl/Software/optfs/
- https://github.com/vijay03/optfs

Evaluation

Evaluation

Is OptFS reliable to random crashes?

- In 400 different random crash scenarios, OptFS proved to be reliable

Evaluation

Is OptFS reliable to random crashes?

- In 400 different random crash scenarios, OptFS proved to be reliable

What is the performance of OptFS?

- Evaluate on different workloads
- Overwrites in OptFS cause 2 writes: one to the journal and one to the file system

Reliability

Reliability

Built a crash-testing framework

Reliability

Built a crash-testing framework

Workloads:

- Append to a file
- Overwrites to an existing file

Reliability

Built a crash-testing framework

Workloads:

- Append to a file
- Overwrites to an existing file

Crash after re-ordering writes

Reliability

Built a crash-testing framework

Workloads:

- Append to a file
- Overwrites to an existing file

Crash after re-ordering writes

Recover from crashed image

Test for consistency

Reliability

Built a crash-testing framework

Workloads:

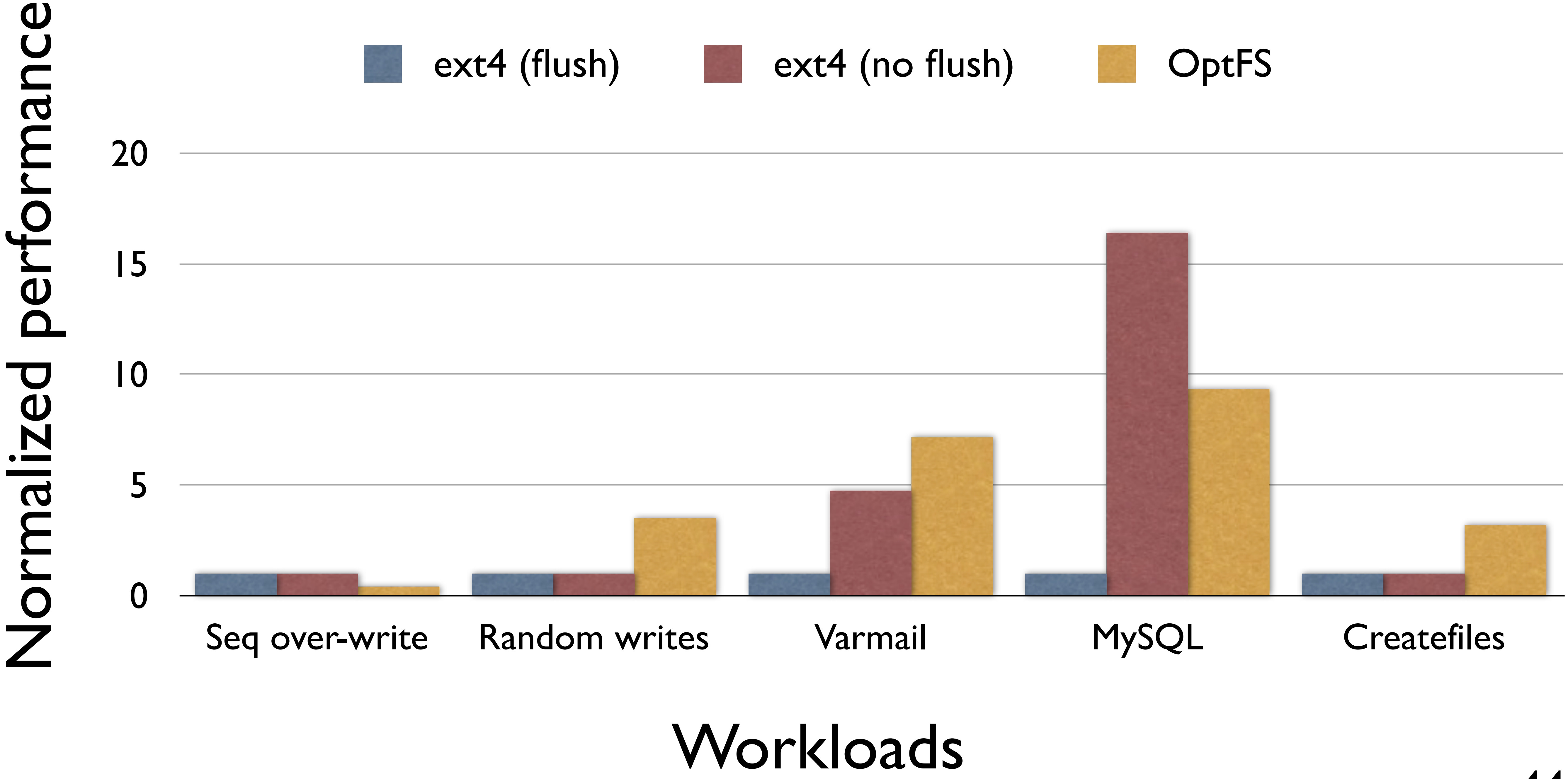
In 400 different crash scenarios, OptFS proved to be reliable

Crash after re-ordering writes

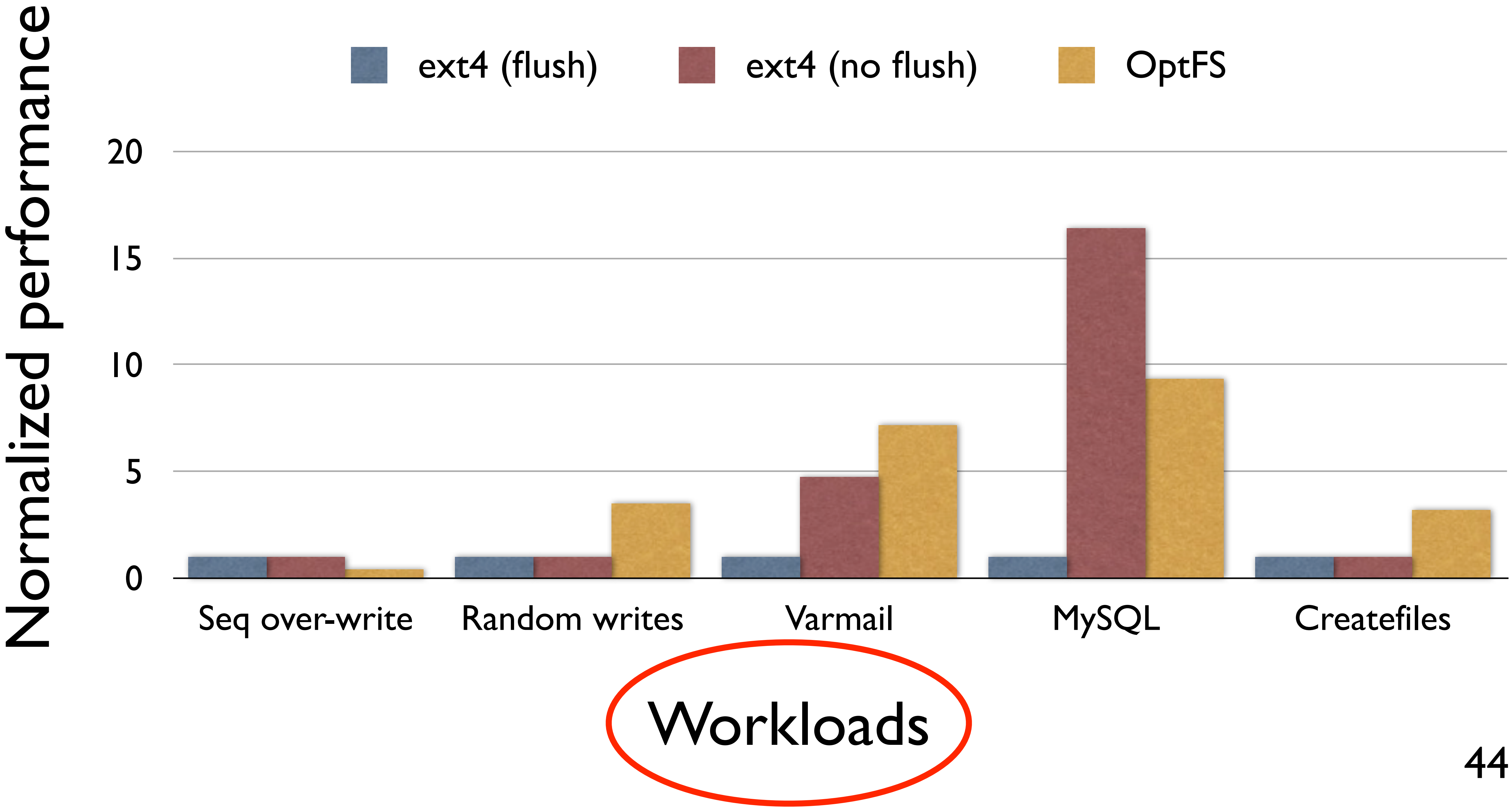
Recover from crashed image

Test for consistency

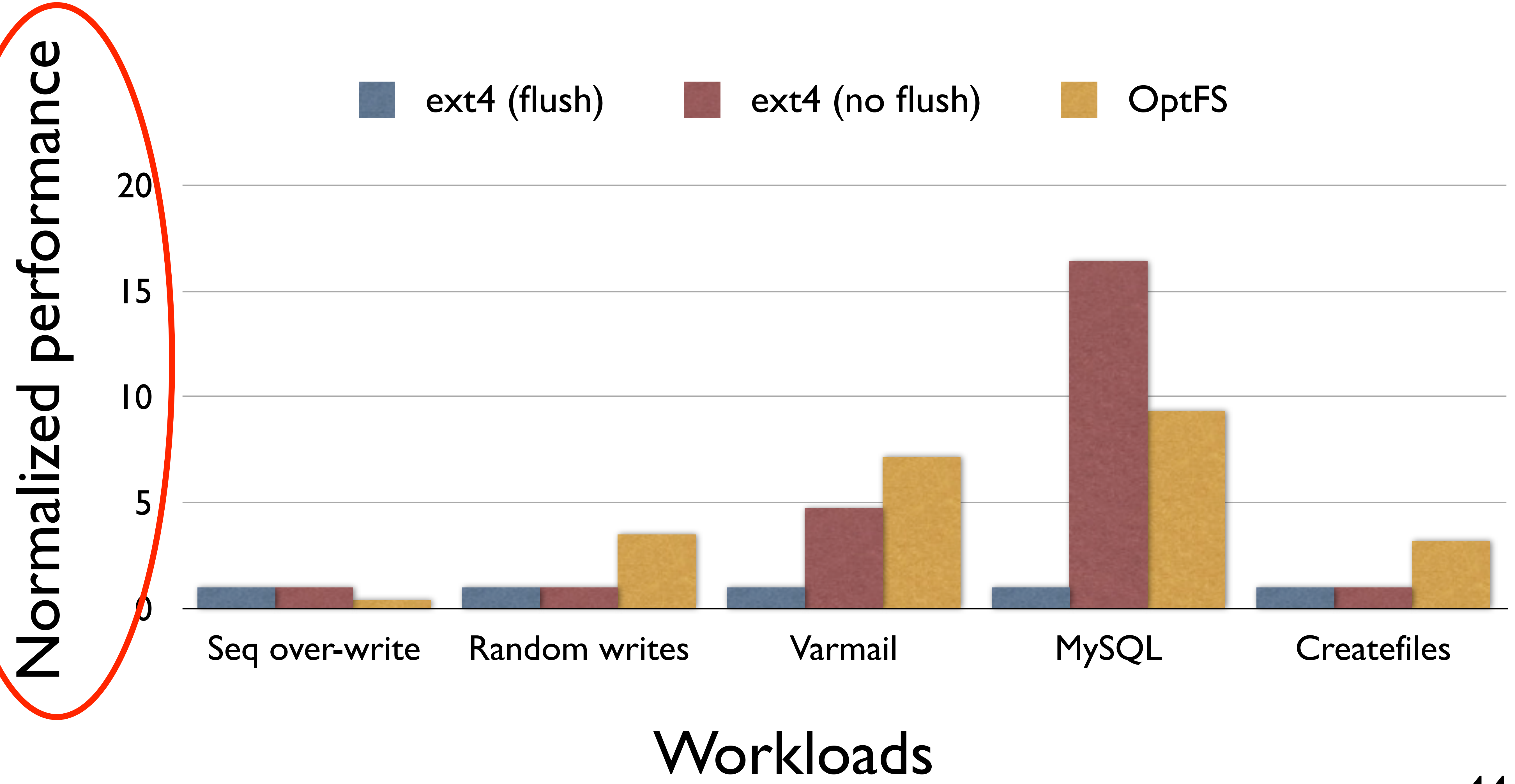
Performance



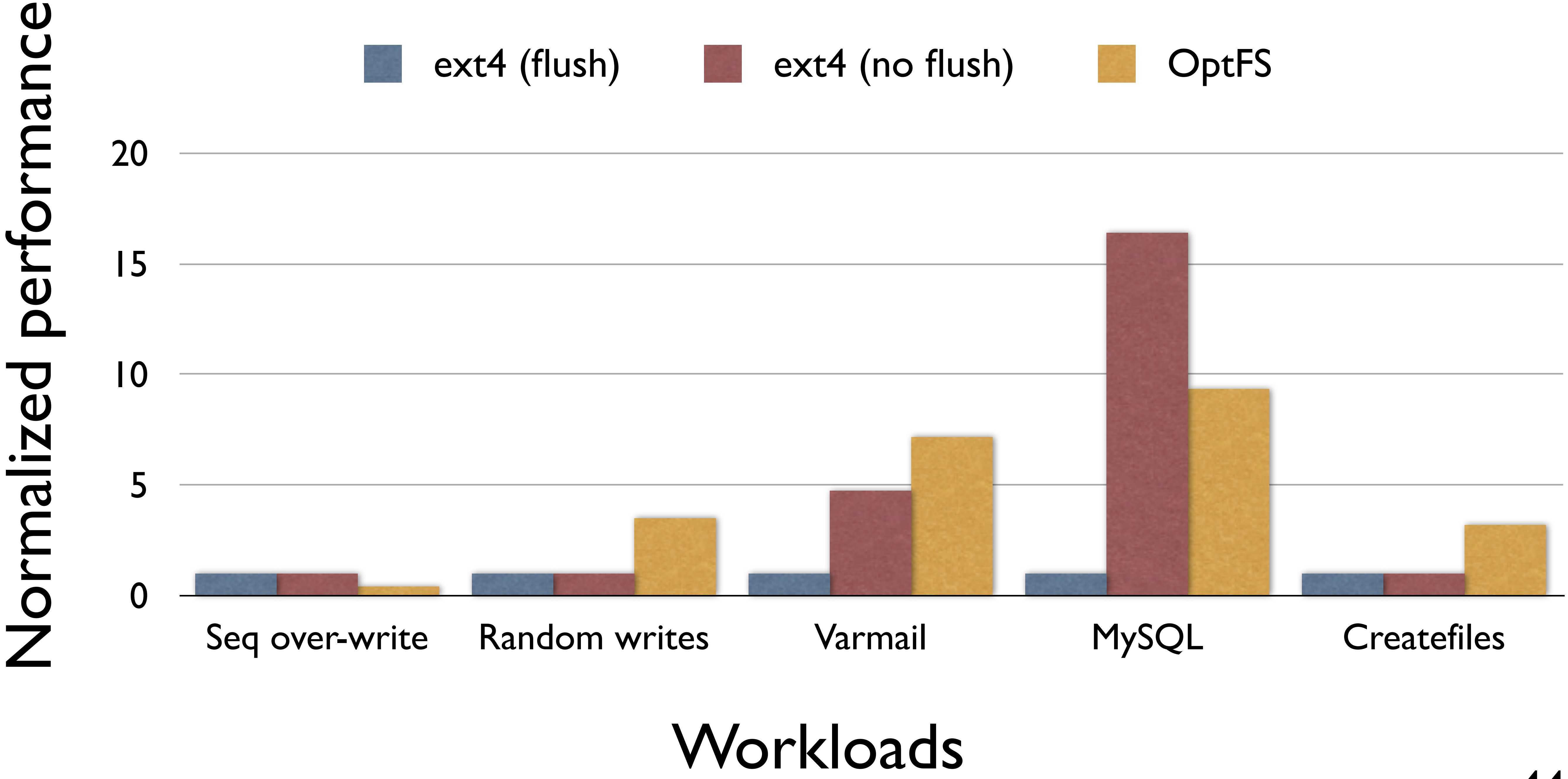
Performance



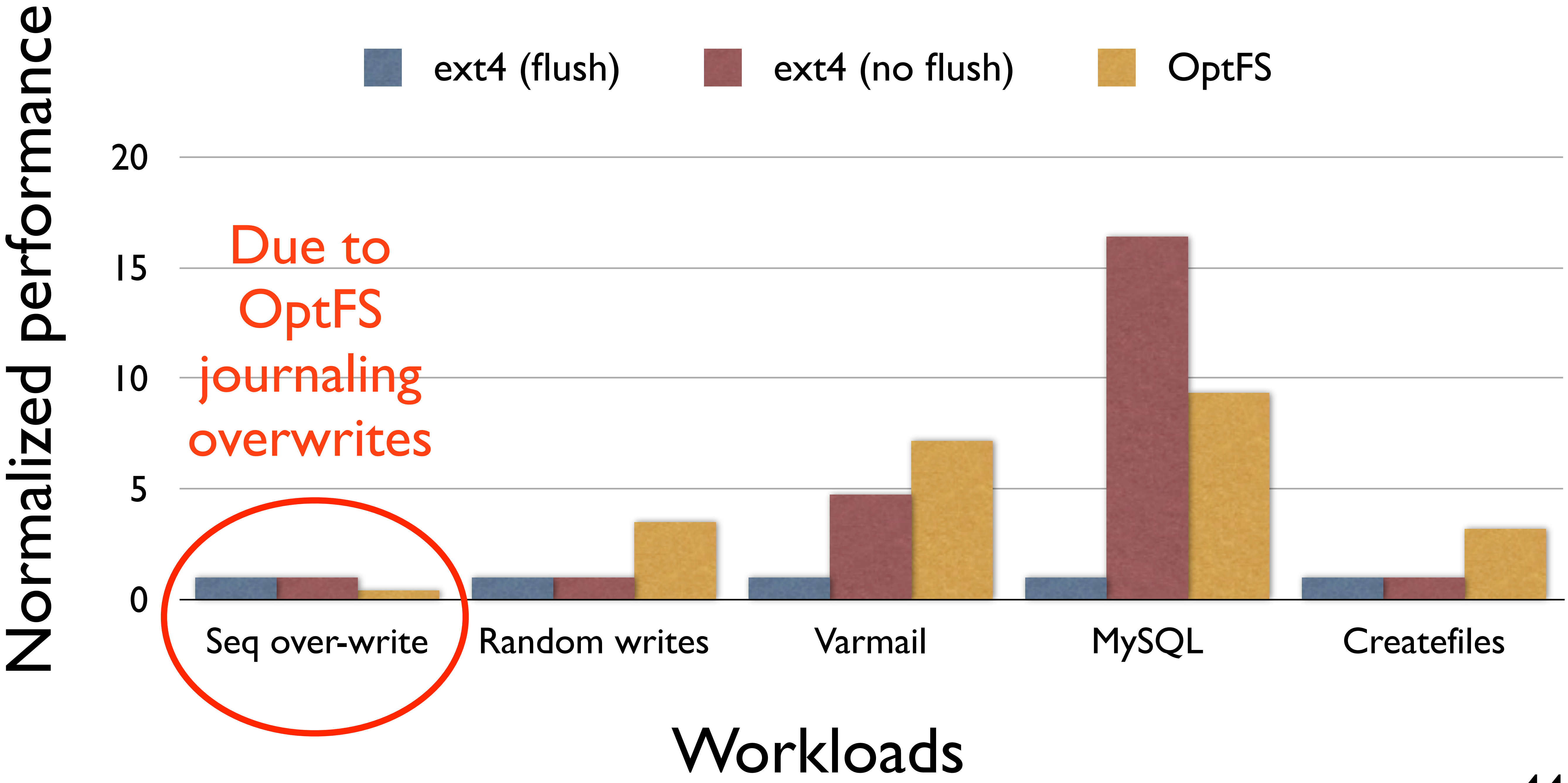
Performance



Performance

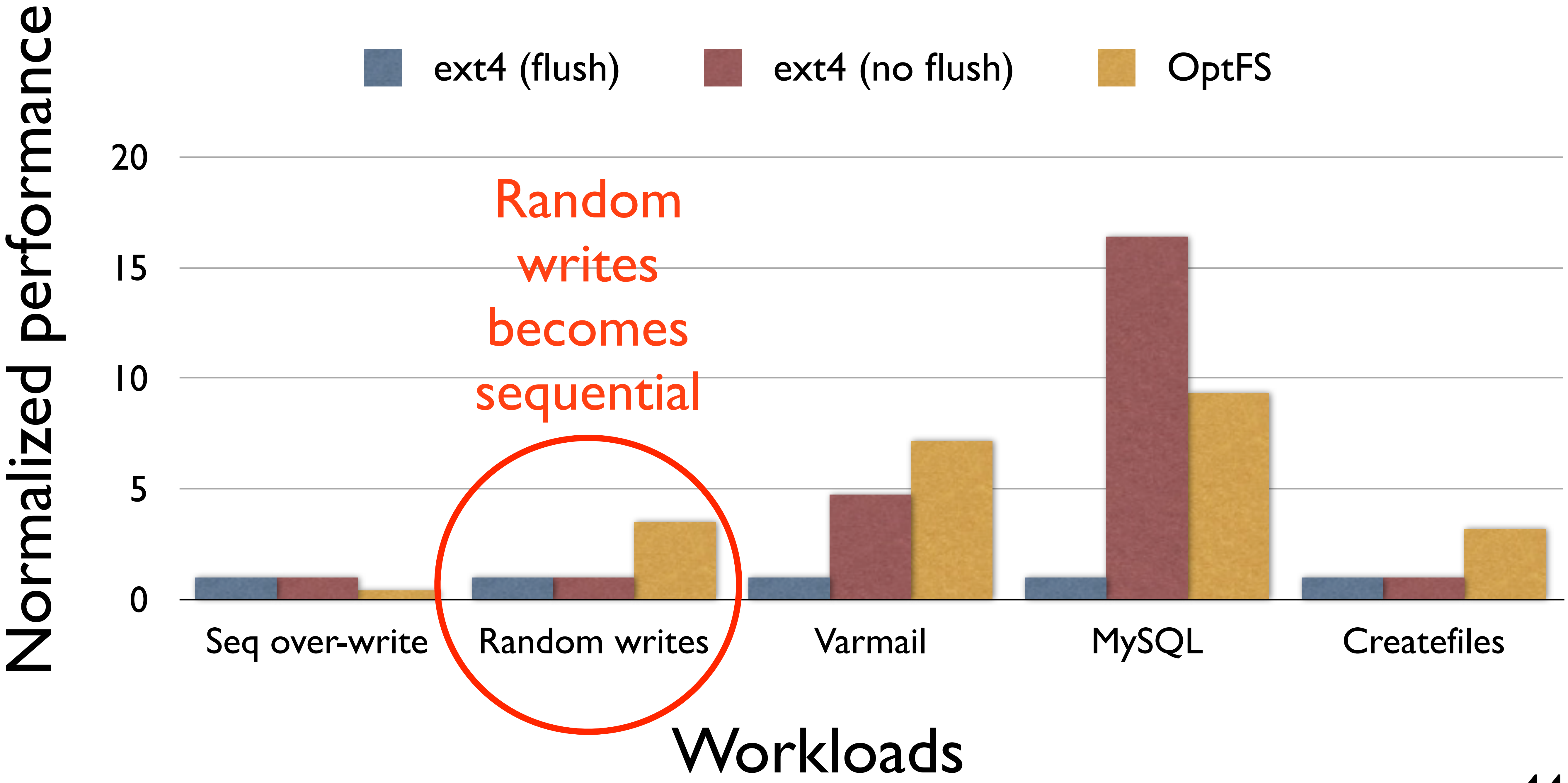


Performance



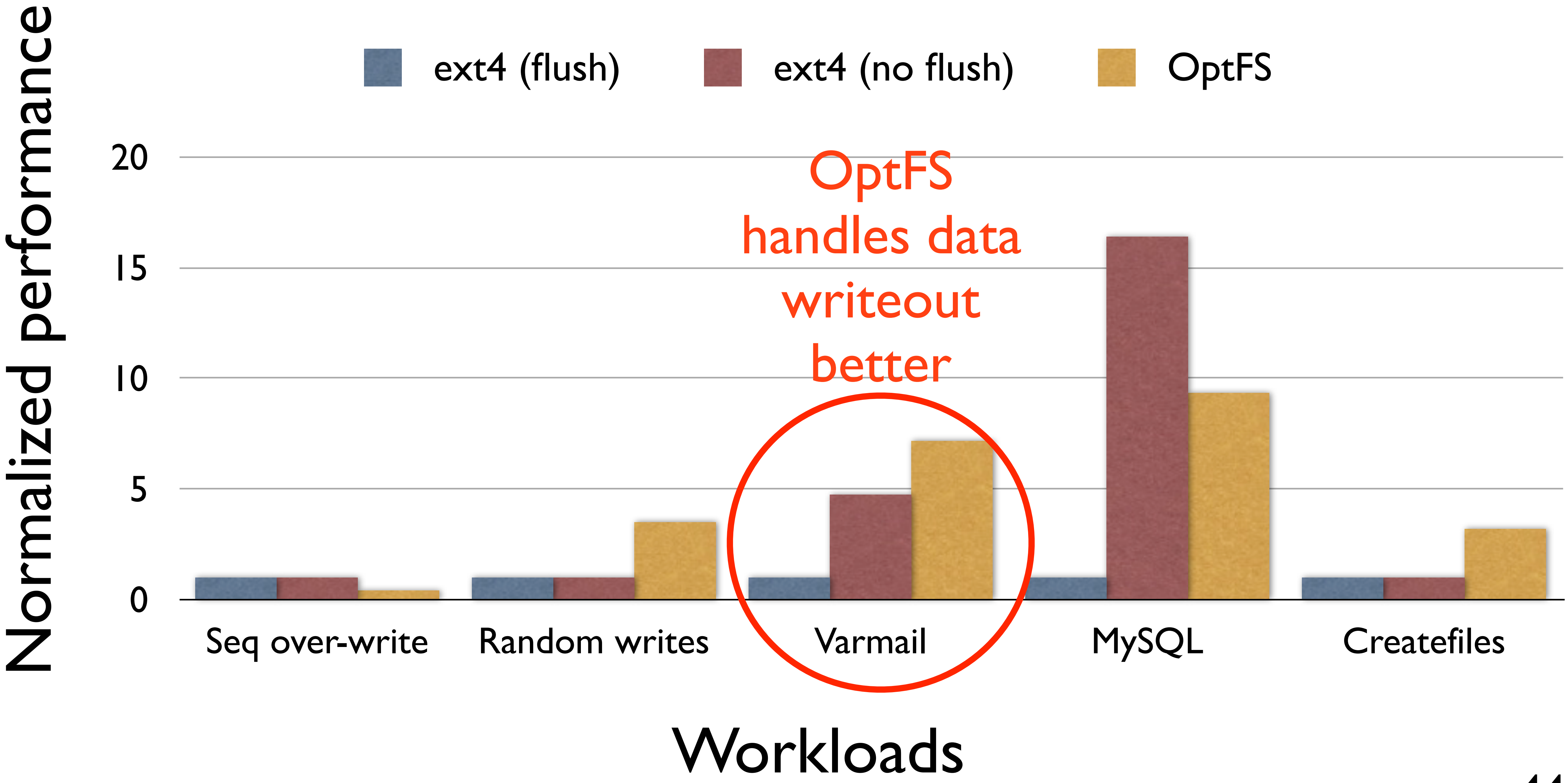
Due to OptFS journaling overwrites

Performance



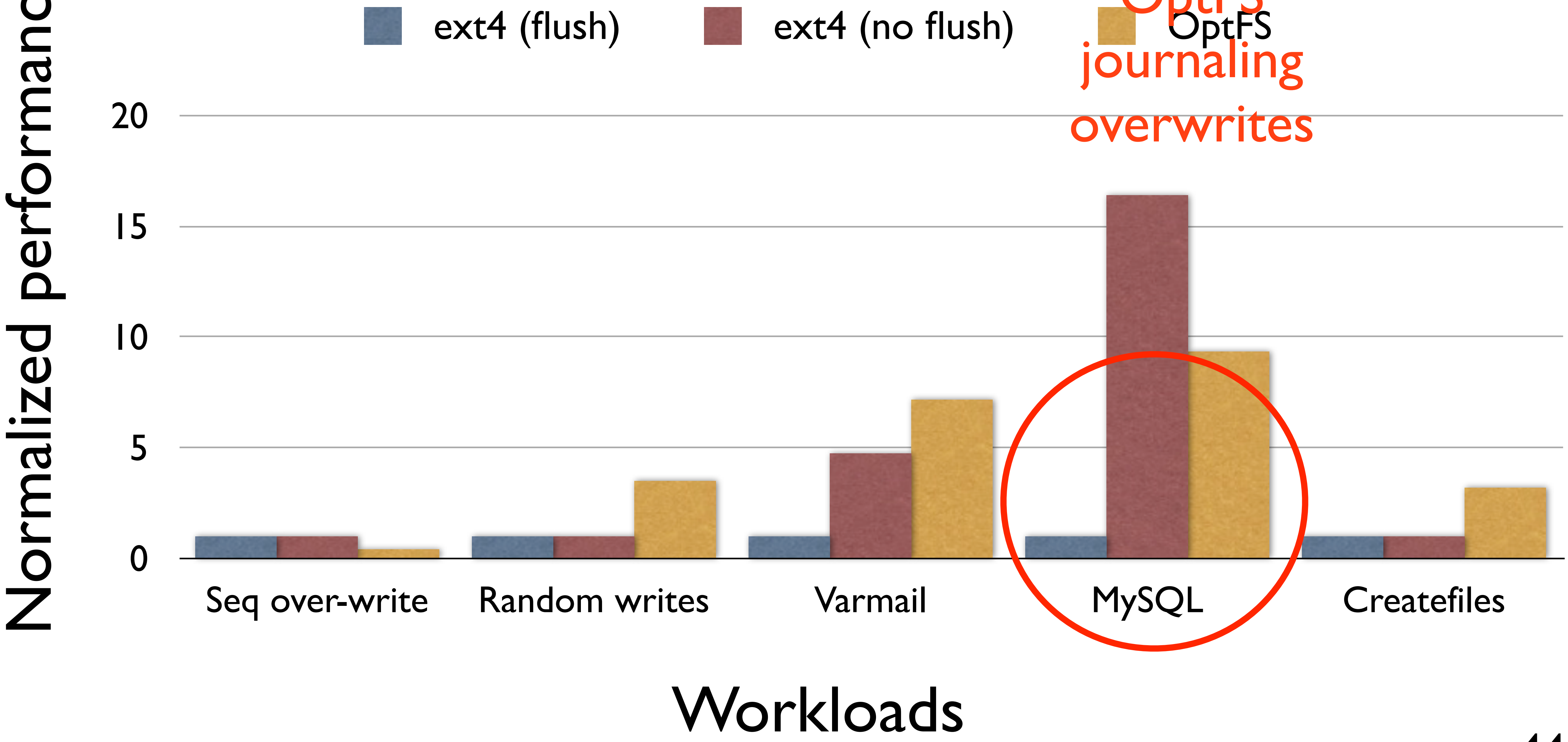
Random writes becomes sequential

Performance



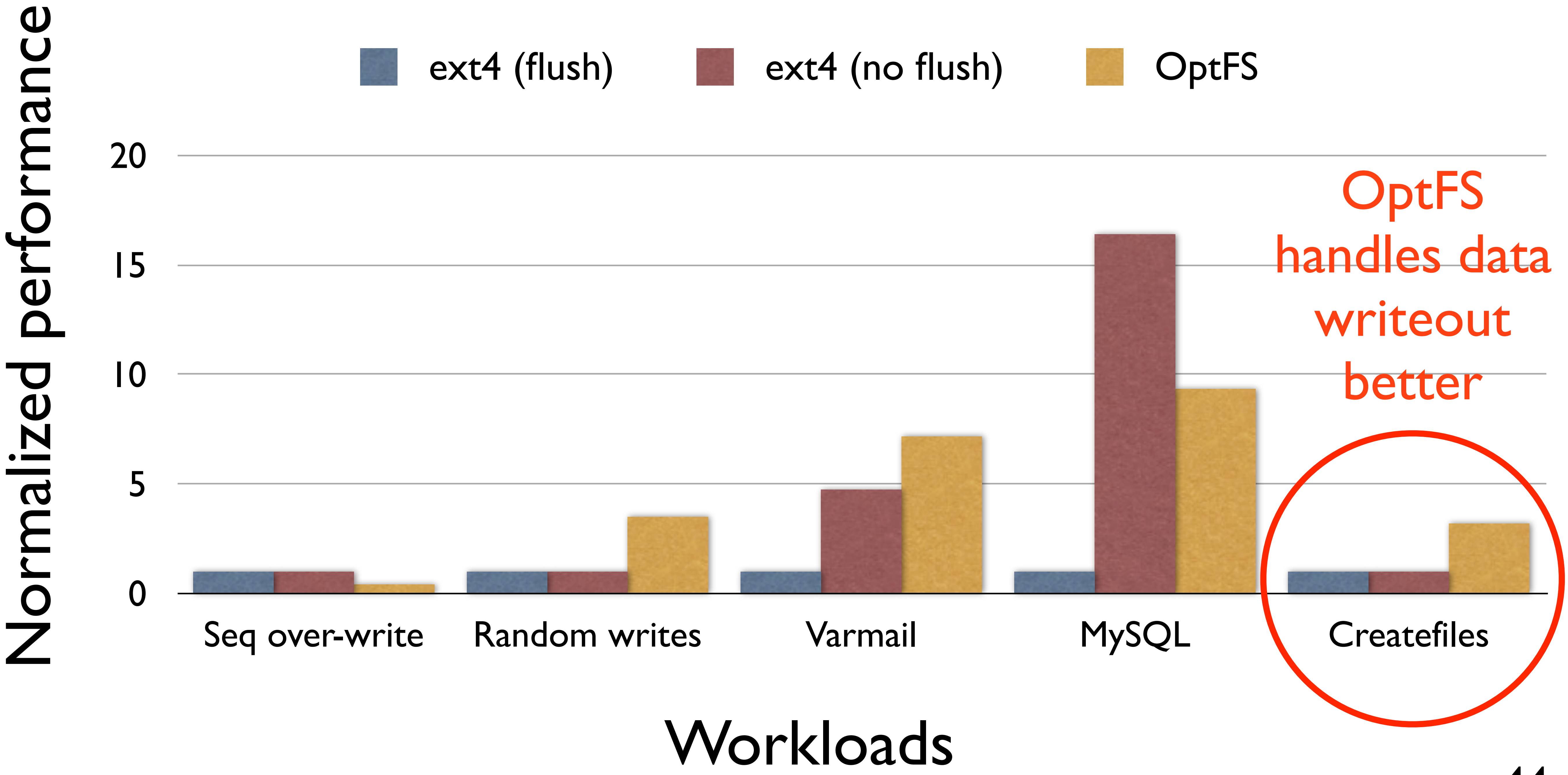
OptFS handles data writeout better

Performance



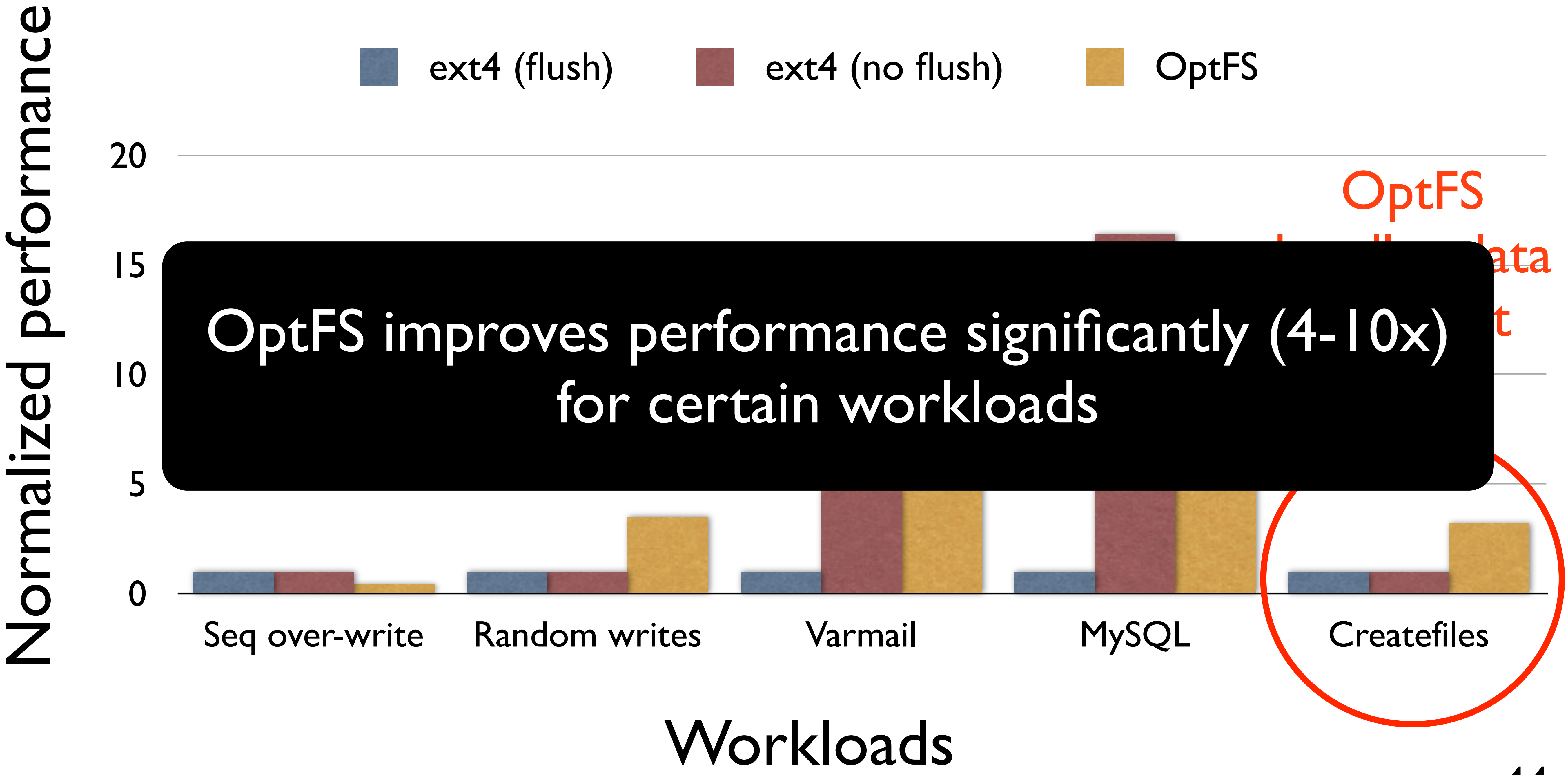
Due to OptFS journaling overwrites

Performance



OptFS handles data writeout better

Performance



OptFS improves performance significantly (4-10x) for certain workloads

Application level consistency

Application level consistency

Can meaningful crash consistency be built on top of OptFS?

Application level consistency

Can meaningful crash consistency be built on top of OptFS?

Replaced `fysnc()` with `osync()`

Application level consistency

Can meaningful crash consistency be built on top of OptFS?

Replaced `fysnc()` with `osync()`

Studied behavior on recovery from random crashes:

- Gedit
- SQLite

Consistency in SQLite

Consistency in SQLite

File-system consistency	*SQLite consistency*

Consistency in SQLite

File-system consistency	*SQLite consistency*
no sync	None

Consistency in SQLite

File-system consistency	*SQLite consistency*
no sync	None
`osync()`	ACI (with eventual durability)

Consistency in SQLite

File-system consistency	*SQLite consistency*
no sync	None
`osync()`	ACI (with eventual durability)
`fsync()`	ACID

Consistency in SQLite

File-system consistency	*SQLite consistency*
no sync	None
`osync()`	ACI (with eventual durability)
`fsync()`	ACID

Crash SQLite in the middle of a transaction

Consistency in SQLite

File-system consistency	*SQLite consistency*
no sync	None
`osync()`	ACI (with eventual durability)
`fsync()`	ACID

Crash SQLite in the middle of a transaction

Experimentally SQLite is consistent, but potentially stale

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

	Ext4 w/o flush	Ext4 w/ flush	OptFS
Inconsistent	73	0	0
Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Case studies: Application level consistency

Application: SQLite

Total crashpoints: 100

SQLite is able to provide ACI semantics with osync(), at 10x performance

Old state	8	50	76
New state	19	50	24
Time per op (ms)	23.28	152	15.3

Conclusion

OptFS provides consistency without flushes

Asynchronous Durability Notifications allow the disk to perform optimally

Eventual Durability trades freshness for increased performance

osync() provides a cheap primitive to order application writes

Project Ideas

1. Delayed Durability
2. OptFS on Flash
3. Optimistic btrfs
4. p-inconsistency for RAID, Flash
5. Rewrite applications with osync()/dsync()
6. Forced Unit Access (FUA) study
7. Consistency testing framework

Project Ideas

If you are interested, come talk to me

vijayc@cs.wisc.edu

7366 CS

Thank you

Questions?

Backup Slides

Resource Consumption

FS	CPU	Memory (MB)
ext4 (flush)	3.39	487
ext4 (no flush)	14.86	516
OptFS	25.32	749

Why not just `fsync()` in the background?

Does not solve the problem for the whole system: flushes will still be caused

Any application using foreground fsync() will be affected

Many mobile applications have auto sync at the same time, causing problems [Agrawal12]

References

[Cipar12] Cipar, James, Greg Ganger, Kimberly Keeton, Charles B. Morrey III, Craig AN Soules, and Alistair Veitch. "LazyBase: trading freshness for performance in a scalable database." In Proceedings of the 7th ACM european conference on Computer Systems, pp. 169-182. ACM, 2012.

[Chidambaram12] Chidambaram, Vijay, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Consistency without ordering." In Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12). 2012.

[Chidambaram13*] Chidambaram, Vijay, Thanumalayan Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Optimistic Crash Consistency." Submitted to SOSP 2013

[Ganger94] Ganger, Gregory R., and Yale N. Patt. "Metadata update performance in file systems." In Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation, p. 5. USENIX Association, 1994.

References

References

[Ghemawat03] Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." In ACM SIGOPS Operating Systems Review, vol. 37, no. 5, pp. 29-43. ACM, 2003.

[Hagmann87] Hagmann, Robert. Reimplementing the Cedar file system using logging and group commit. Vol. 21, no. 5. ACM, 1987.

[Hitz94] Hitz, Dave, James Lau, and Michael Malcolm. "File system design for an NFS file server appliance." In Proceedings of the USENIX Winter 1994 Technical Conference, pp. 235-246. 1994.

[Keeton04] Keeton, Kimberley, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. "Designing for disasters." In Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp. 59-62. 2004

References

References

[Lamport98] Lamport, Leslie. "The part-time parliament." ACM Transactions on Computer Systems (TOCS) 16, no. 2 (1998): 133-169.

[Lamport01] Lamport, Leslie. "Paxos made simple." ACM SIGACT News 32, no. 4 (2001): 18-25

[McKusick84] McKusick, Marshall K., William N. Joy, Samuel J. Leffler, and Robert S. Fabry. "A fast file system for UNIX." ACM Transactions on Computer Systems (TOCS) 2, no. 3 (1984): 181-197.

[Ongaro2013] Ongaro, Diego, and John Ousterhout. "In Search of an Understandable Consensus Algorithm." 2013.

References

References

[Ousterhout10] Ousterhout, John, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra et al. "The case for RAMClouds: scalable high-performance storage entirely in DRAM." ACM SIGOPS Operating Systems Review 43, no. 4 (2010): 92-105.

[Prabhakaran05] Prabhakaran, Vijayan, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. Vol. 39, no. 5. ACM, 2005

[Rosenblum92] Rosenblum, Mendel, and John K. Ousterhout. "The design and implementation of a log-structured file system." ACM Transactions on Computer Systems (TOCS) 10, no. 1 (1992): 26-52.

[Shvachko10] Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system." In Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, pp. 1-10. IEEE, 2010.