2025년 10월 23일

NVRAMOS'25

제주 신라호텔

Preserving the Order in Modern IO Stack

KAIST

Youjip Won

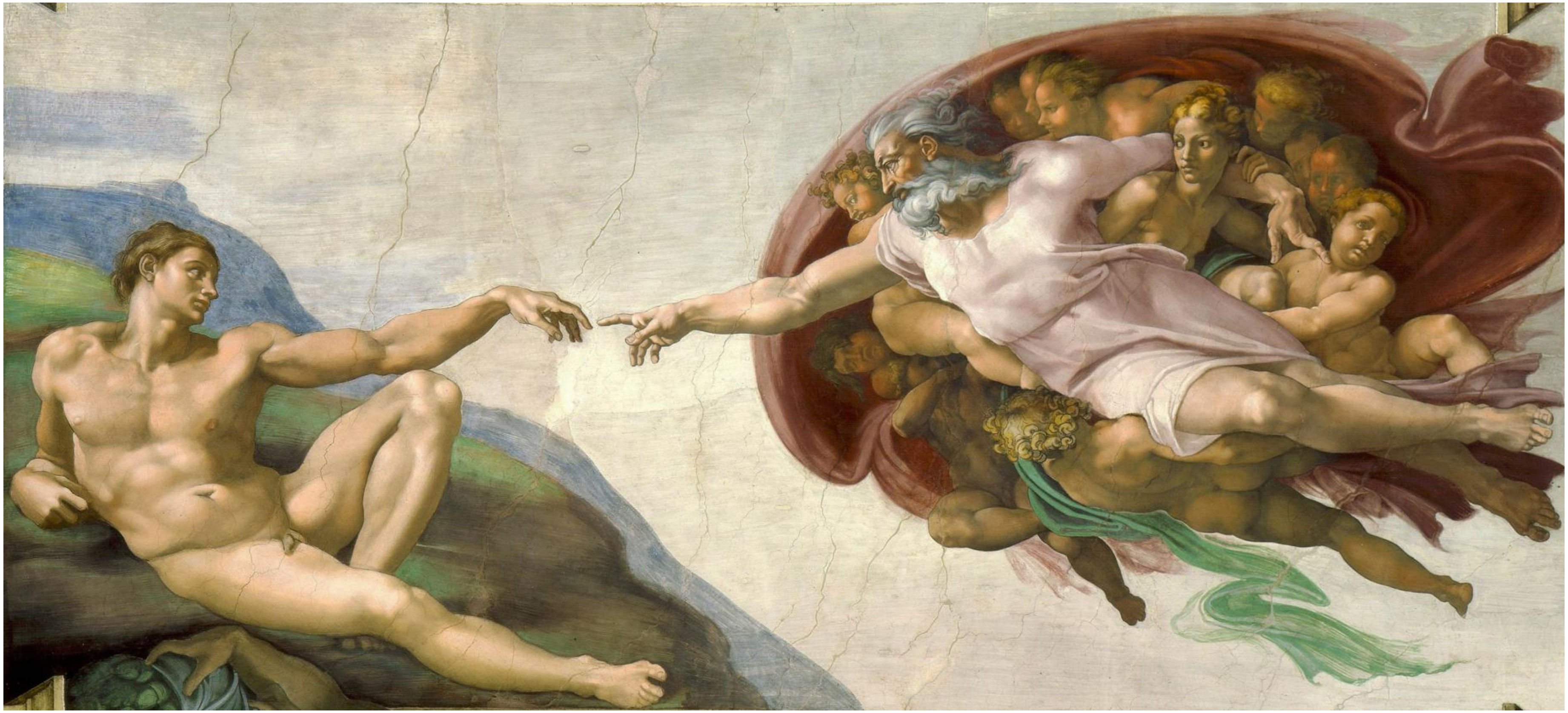
Outline

1. Background
2. Storage Order in single queue IO stack
3. Storage Order and concurrency
4. Storage Order in multi queue IO stack
5. Conclusion

Chaos,
The Genesis

Ivan Aivazovsky
1831, Oil Canvas

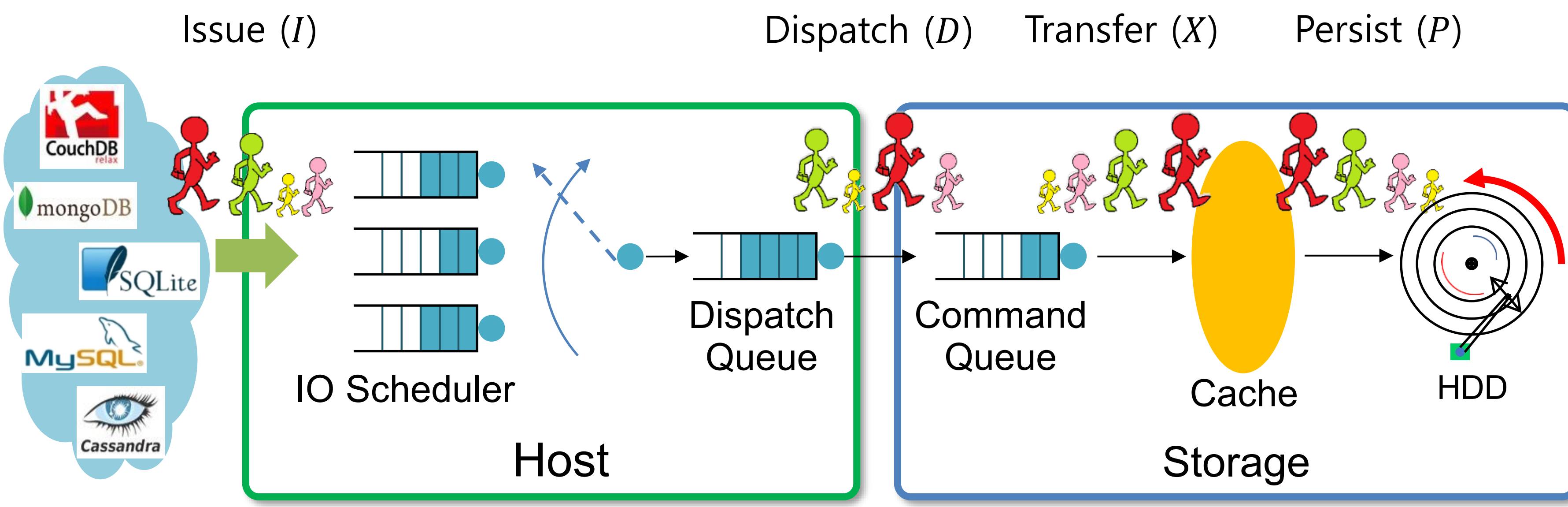
Orders



The Creation of Adam, a fresco painting,
Michelangelo, 1508–1512

Modern IO Stack

Modern IO stack is Orderless.



$I \neq D$: IO Scheduling

$D \neq X$: Time out / retry

$X \neq P$: Cache replacement, page table update algorithm of FTL

Storage Order

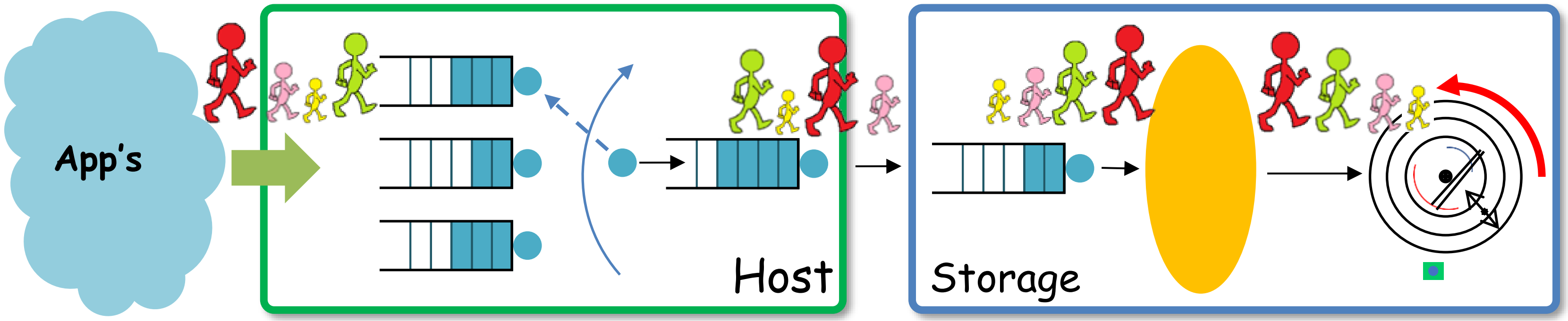
The order in which the data blocks are made durable.

Storage order guarantee

Issue (I) ⟷ Persist (P)

$$(I = D) \land (D = X) \land (X = P)$$

Issue (I) ⟷ Dispatch (D) ⟷ Transfer (X) ⟷ Persist (P)

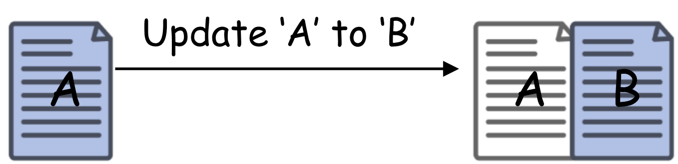


Crash Consistency

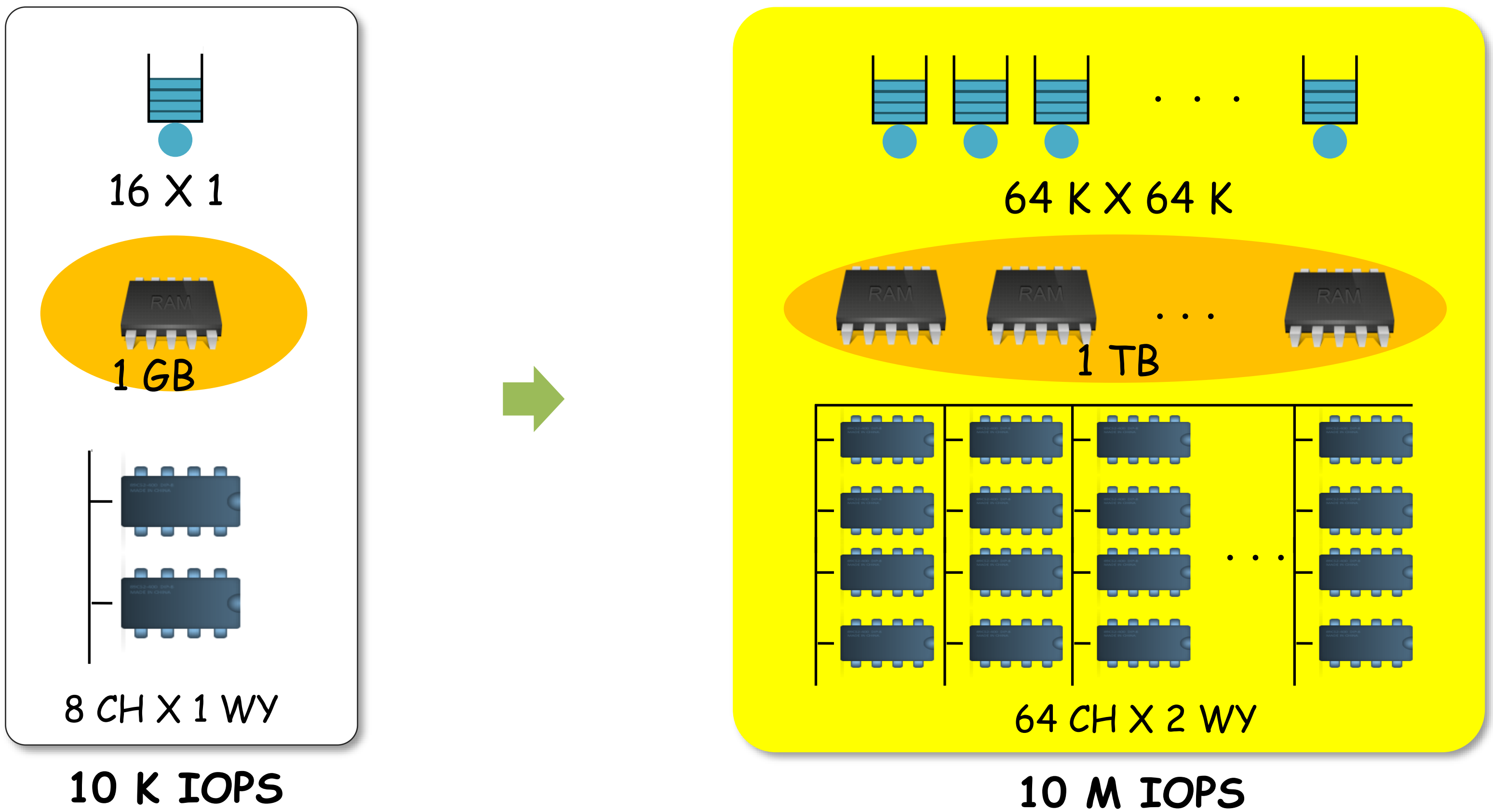
Controlling the storage order is for crash consistency.

Database logging and Filesystem journaling (SQLite, EXT4, RocksDB)

Out-of-place update (BTRFS, F2FS)



Storage Evolution



Intel X25-M
35 K IOPS
2009

830 PRO
80 K IOPS
2012

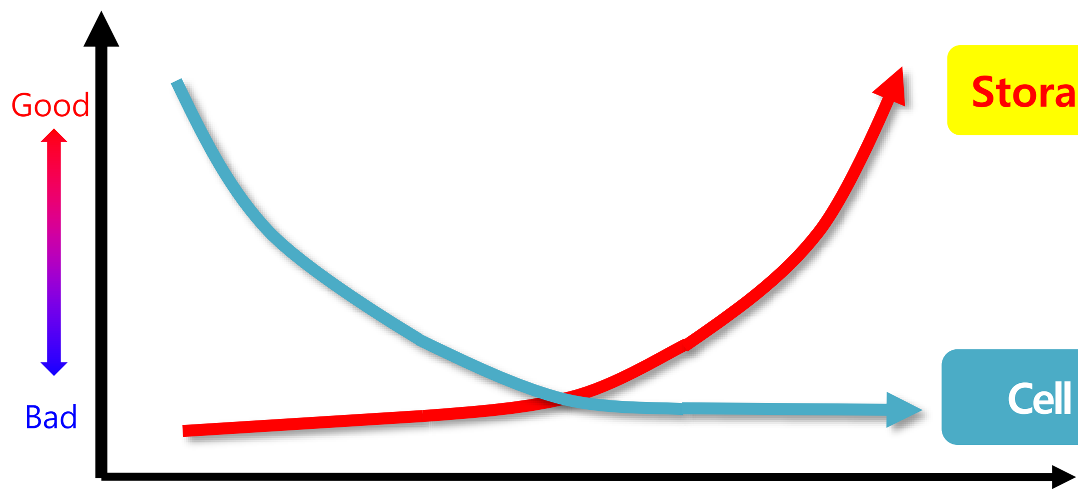
850 PRO
100 K IOPS
2014

Intel 600p
155 K IOPS
2016

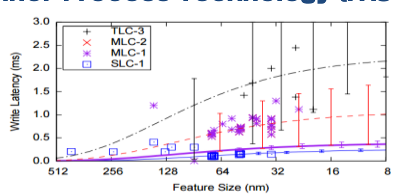
960 PRO
380 K IOPS
2016

PM1725
1 M IOPS
2015

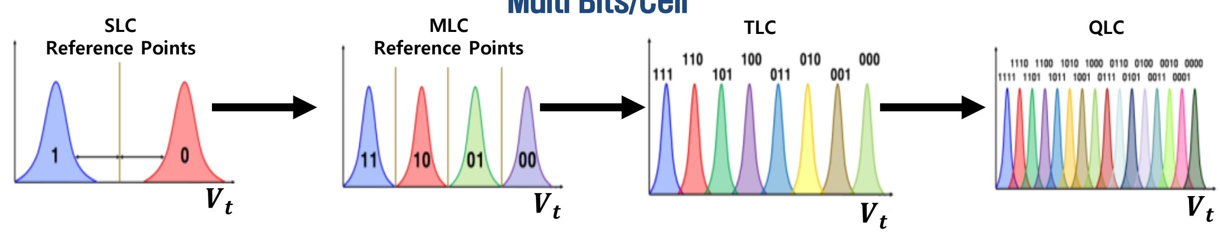
PCI Gen4
10 M IOPS
2022



Finer Process Technology (FAST12)



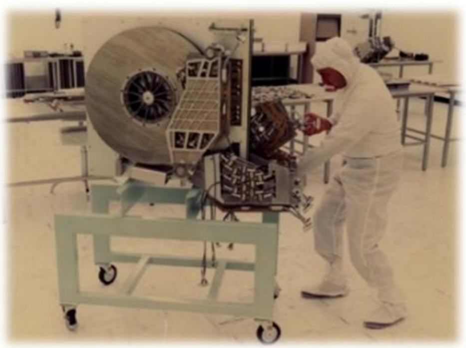
Multi Bits/Cell



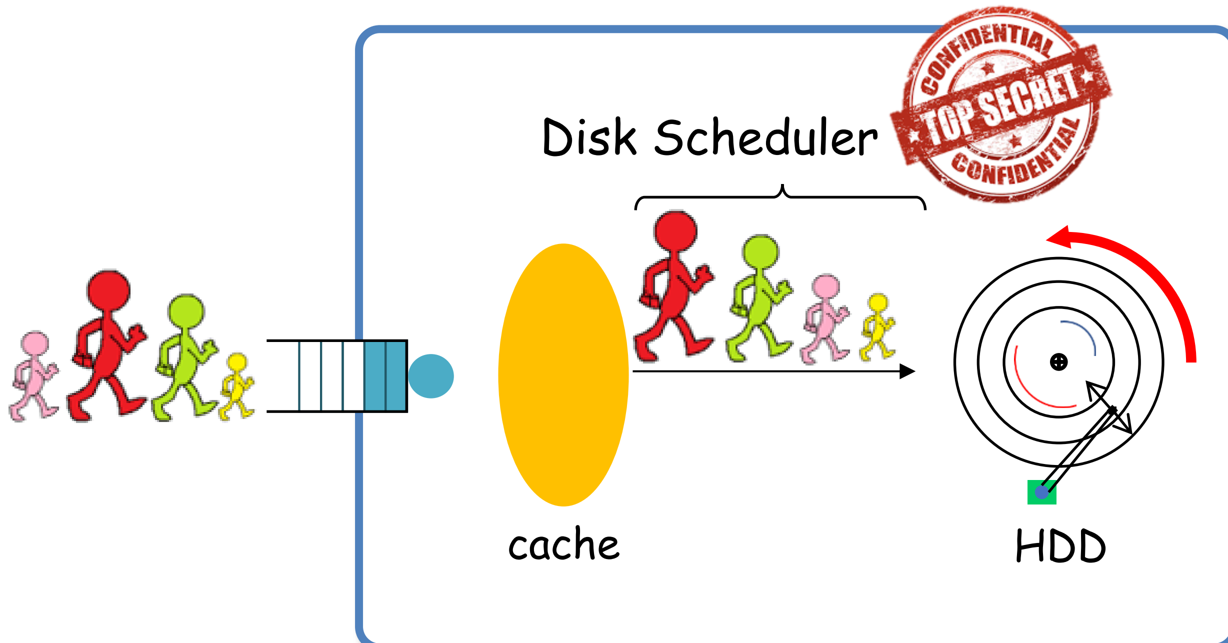
Why has IO stack been orderless for the last 50 years?

In HDD, host cannot control the persist order.

$$(I \cancel{=} P) \equiv (I = D) \wedge (D = X) \wedge (X \cancel{=} P)$$



250MB @ 1970's



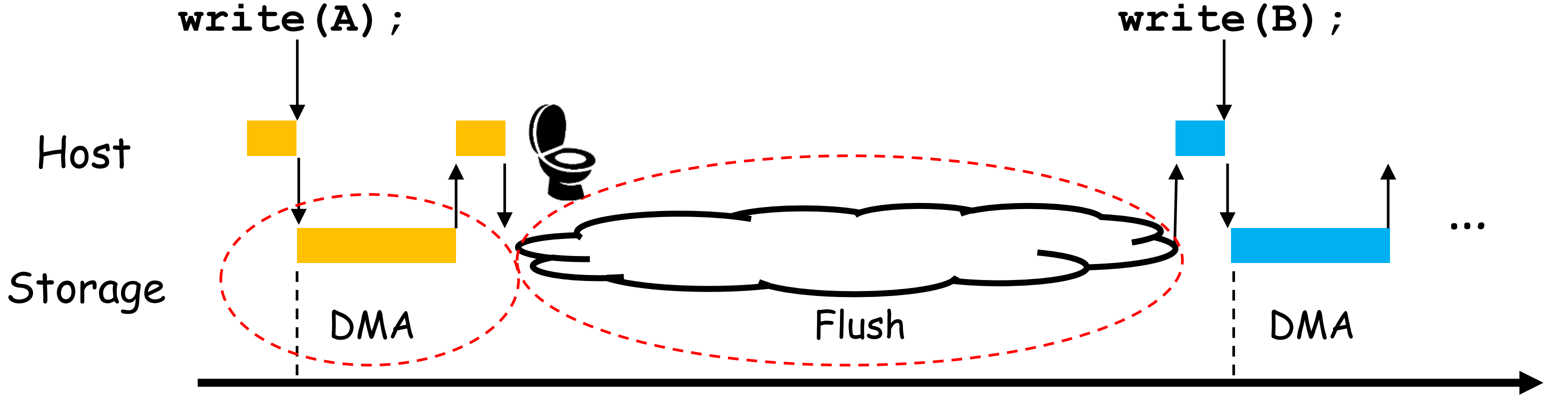
Enforcing Storage Order in Orderless IO Stack

Transfer-and-Flush

```
write (A) ;
write (B) ;
```

→

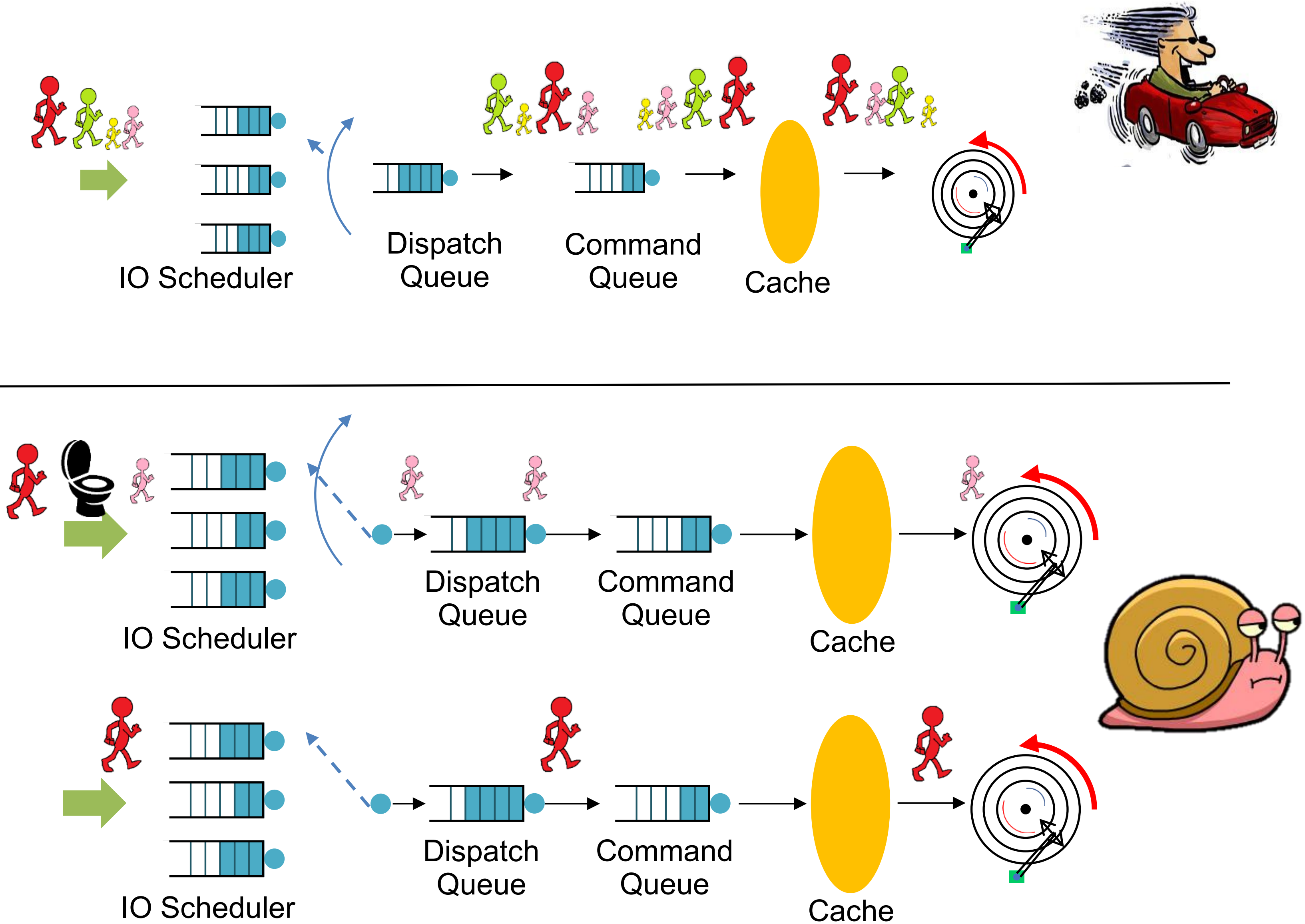
```
write (A) ;
Transfer-and-flush;
write (B) ;
```



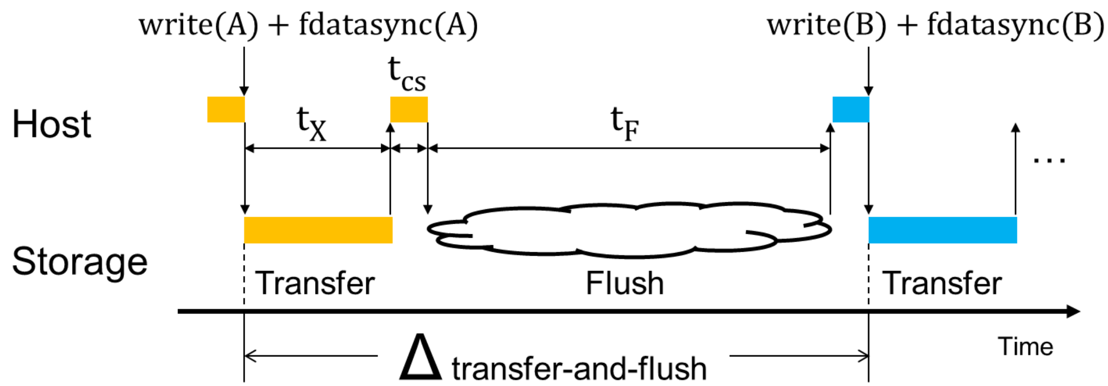
To enforce transfer order, block the caller !

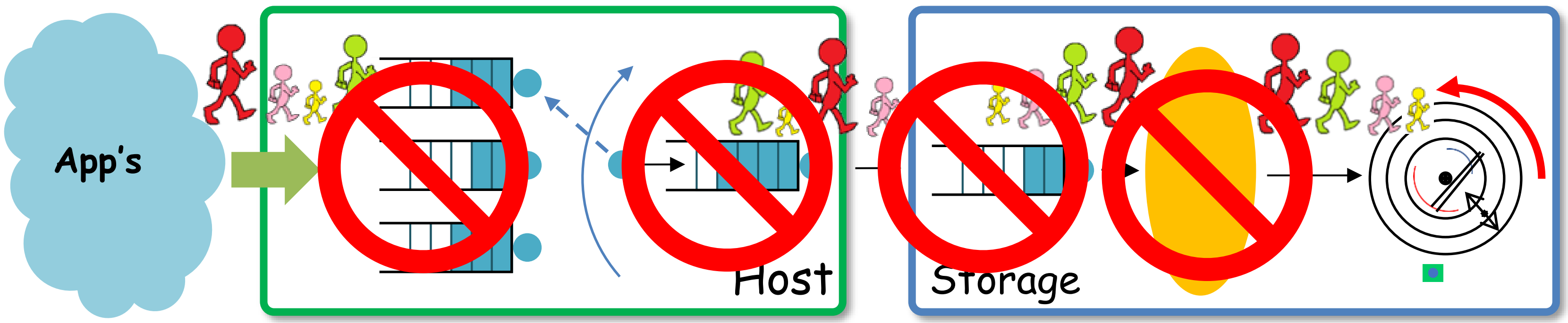
To enforce persist order, drain the cache !

Enforcing Storage Order in Orderless IO Stack

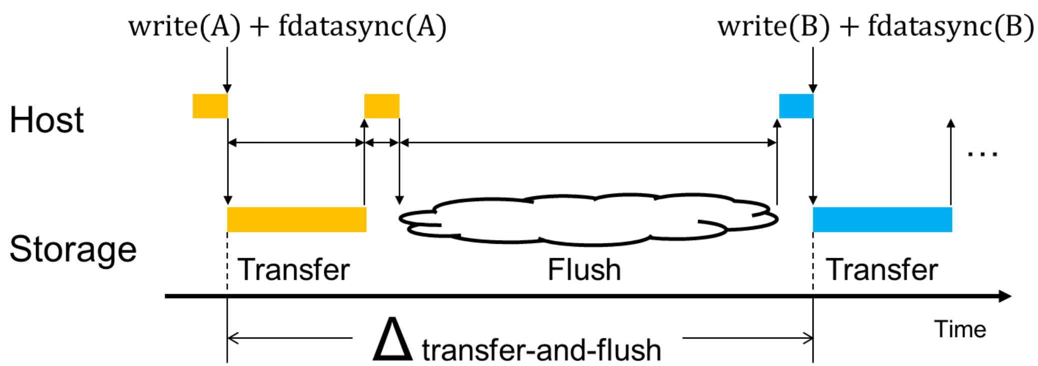


Transfer-and-Flush





Transfer-and-Flush



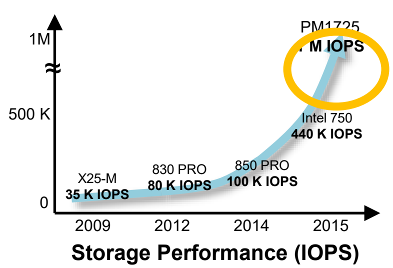
~~Multi Command Queue~~

~~Multi Channel Multi Way~~

~~Disk Cache~~

~~I/O Scheduler~~

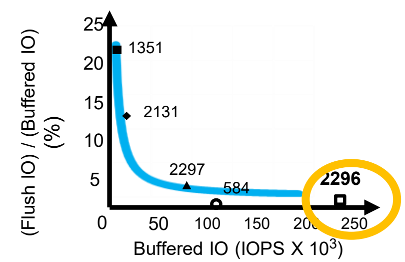
NVMe PM1725
120K IOPS

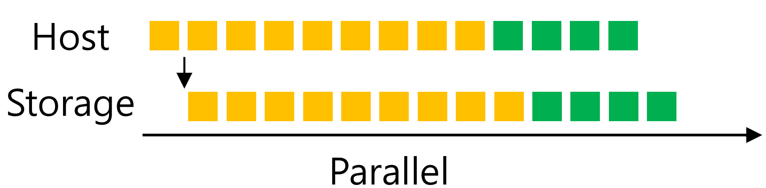


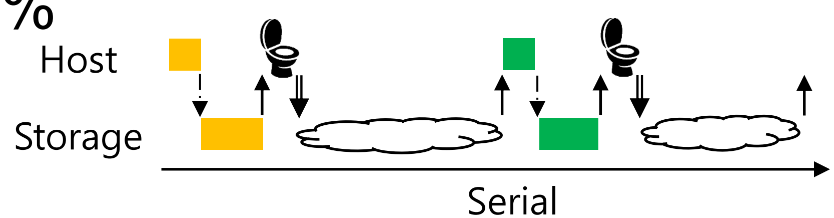
Ordering Guarantee

< 2%

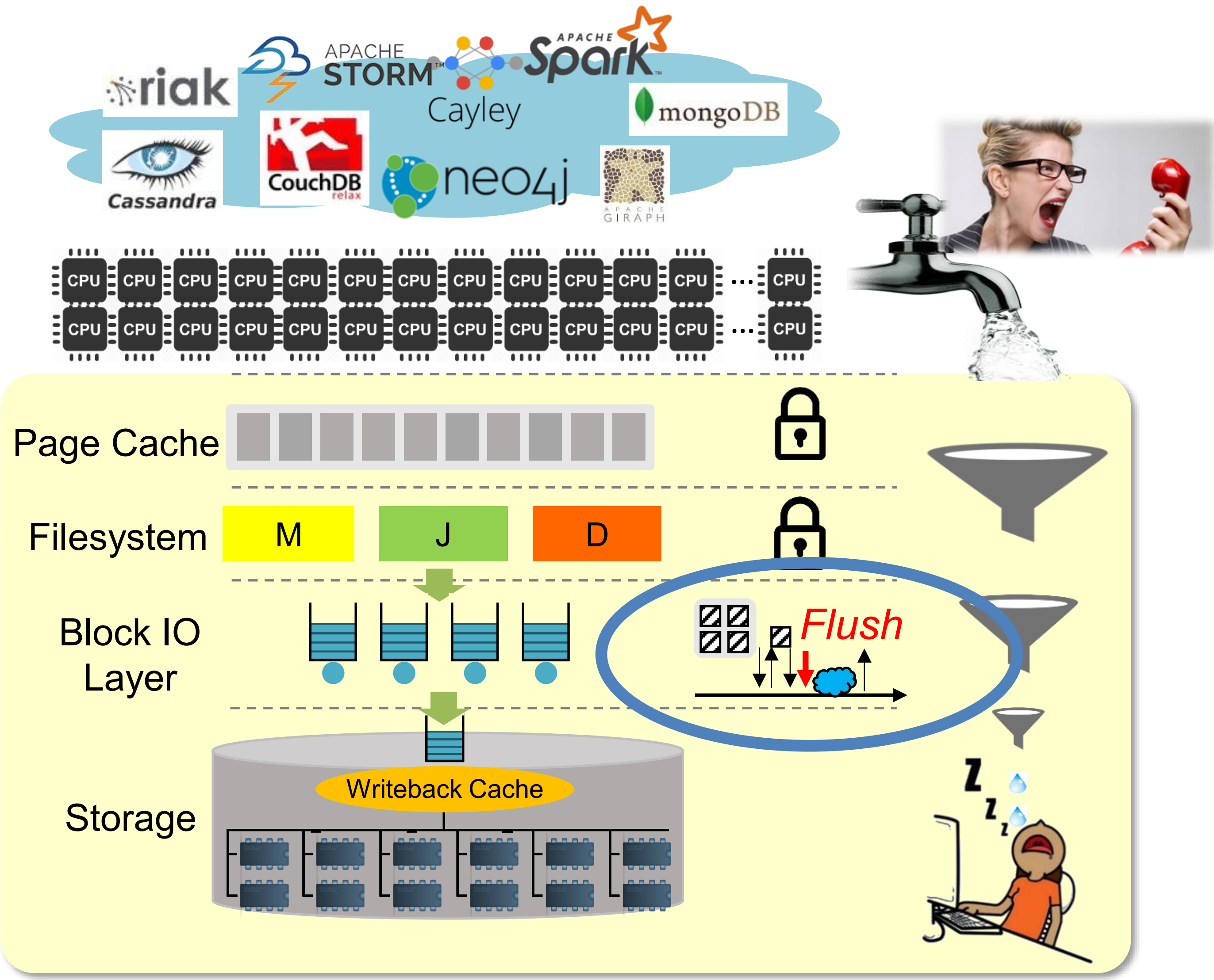
NVMe PM1725
2K IOPS







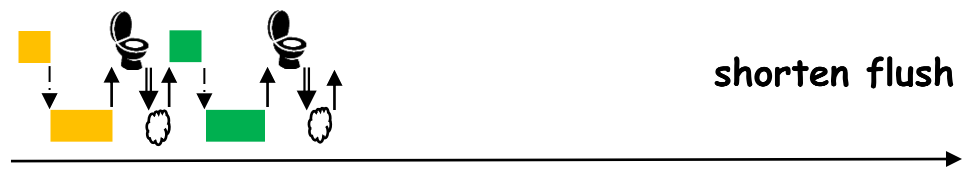
Storage is severely under-utilized.



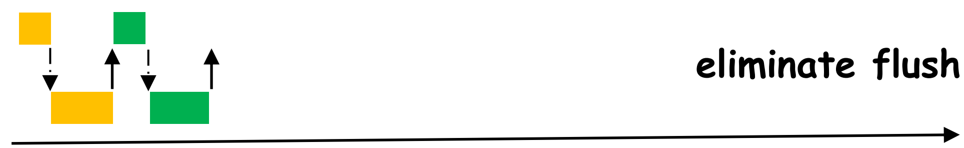
How to mitigate the overhead of storage order guarantee?

Mainly to hide the overhead of transfer-and-flush.

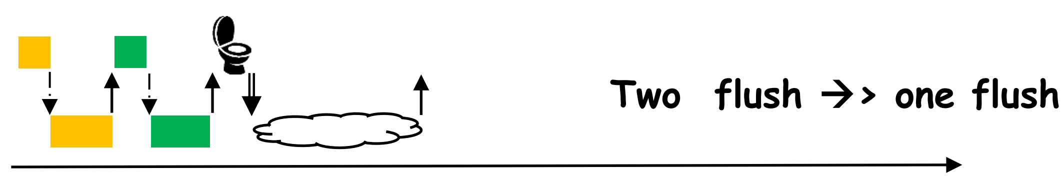
Supercap at SSD



`no_barrier` **mount option (EXT4)**



`journal async commit (EXT4/Android)`



How to mitigate the overhead of storage order guarantee?

- ✓ FeatherStitch [SOSP'07] , NoFS[FAST'12], OptFS[SOSP'13]
  - ➢ HDD, still use flush

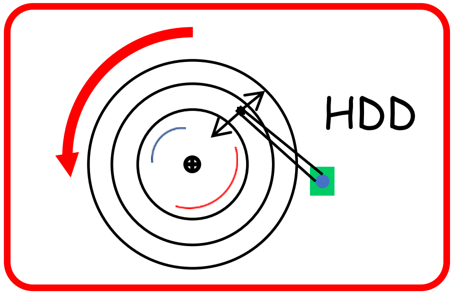
- ✓ HORAE [OSDI'20], ccNVMe [SOSP'21], RIO [EUROSYS'23]
  - ➢ Ordered recovery
  - ➢ On-SSD NVM logging
  - ➢ Multi-queue support → place the ordered IO at the same queue.

- ✓ LazyBarrier[ASPLOS'24]
  - ➢ Ordered IO in Smartphone

Outline

How to mitigate the overhead of storage order guarantee?

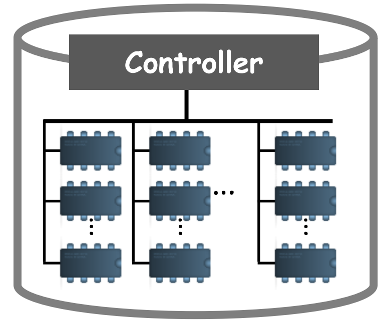
In the era of HDD
(circa 1970)



Seek and rotational delay.

- The host cannot control persist order.
- the IO stack becomes orderless.
- **use transfer-and-flush** to control the storage order

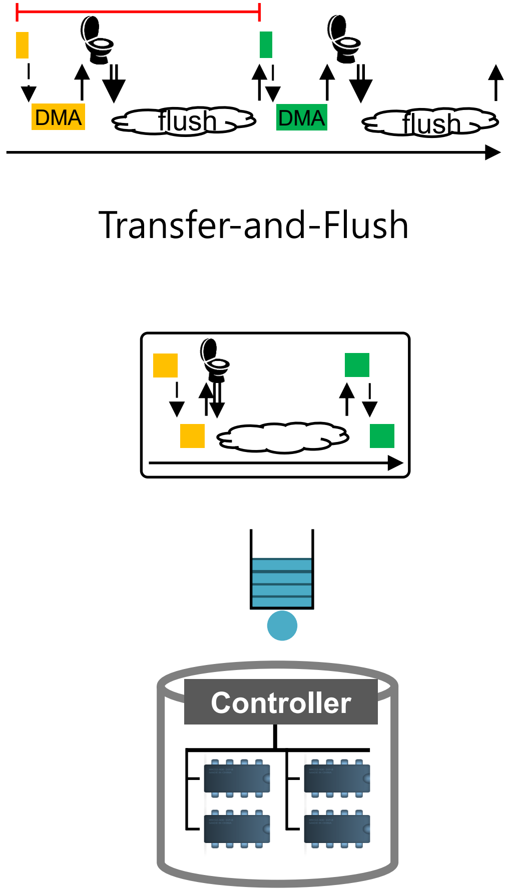
In the era of SSD
(circa 2000)



~~Seek and rotational delay~~

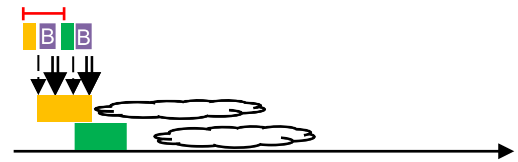
- The host may control persist order.
- The IO stack may become order-preserving.
- Control the storage order **without Transfer-and-Flush**

Barrier-enabled IO Stack (FAST'18)

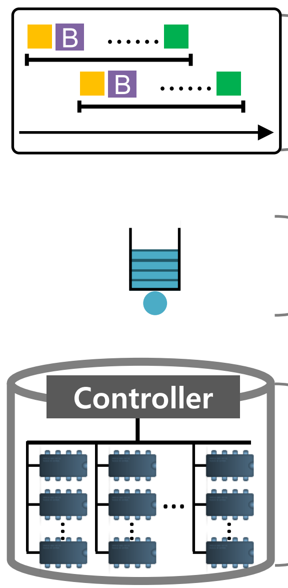


Transfer-and-Flush

Legacy IO Stack



Barrier



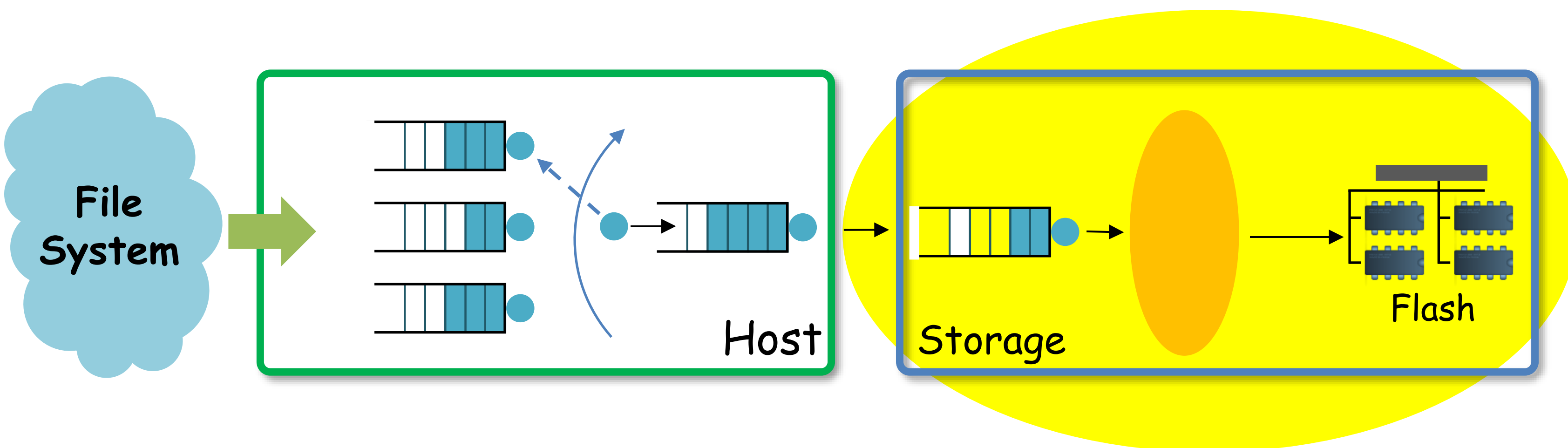
Barrier-enabled
- Dual-Mode Journaling
- fbarrier() / fdatabarrier()

Order-preserving Block dev
- Order-preserving dispatch
- Epoch-based IO scheduling

Order-Preserving Storage
- Barrier write command

Barrier-enabled IO Stack

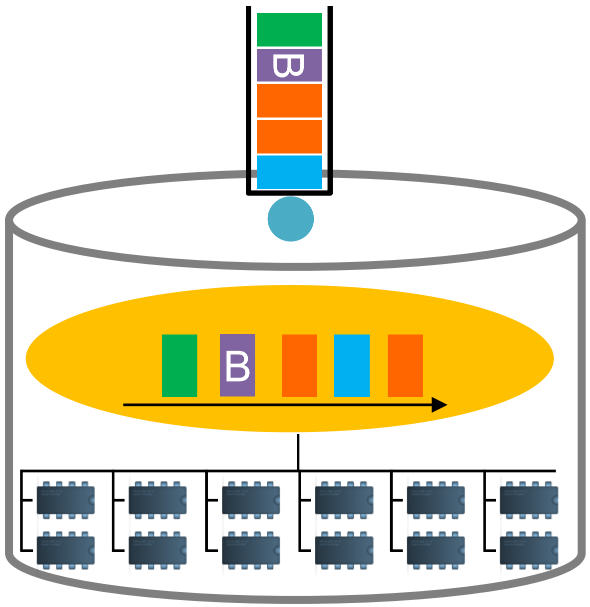
Order-preserving Storage



Satisfying X = P

`barrier` command (2005, eMMC)

```
write (A) ;
barrier;
write (B) ;
write (C) ;
write (D) ;
```



With barrier, Host can control the persist order, $X = P$.

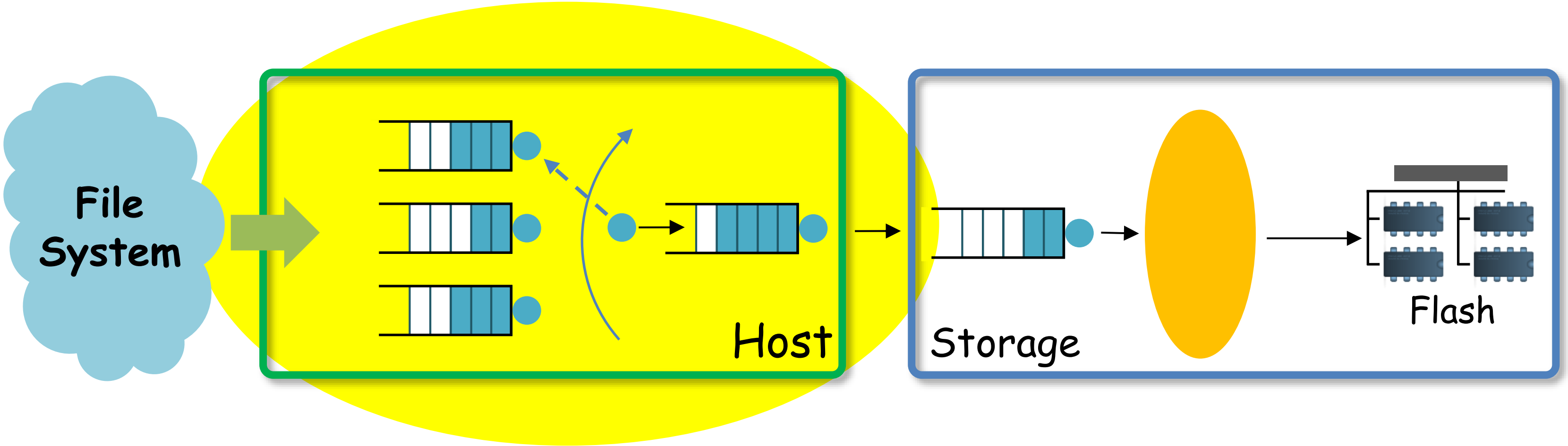
With Barrier command,

host can control the persist order without flush.

$$(I \cancel{=} P) \equiv (I \cancel{=} D) \wedge (D \cancel{=} X) \wedge (X \cancel{=} P)$$

`cache-barrier` was defined at 2005.

Order-preserving Block Layer

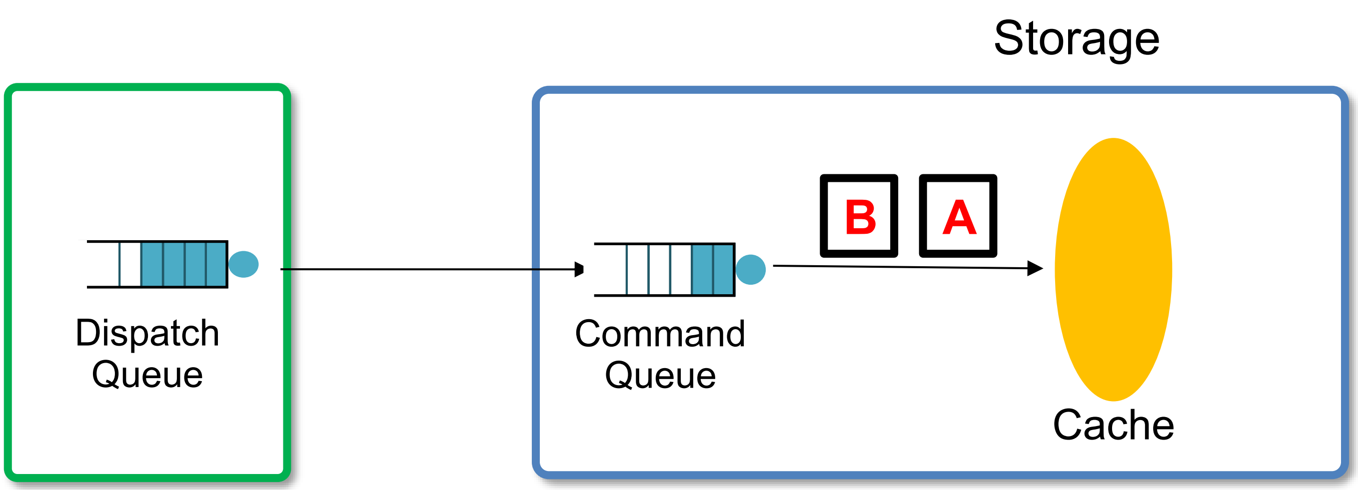


Satisfying D = X

✓ Order Preserving Dispatch

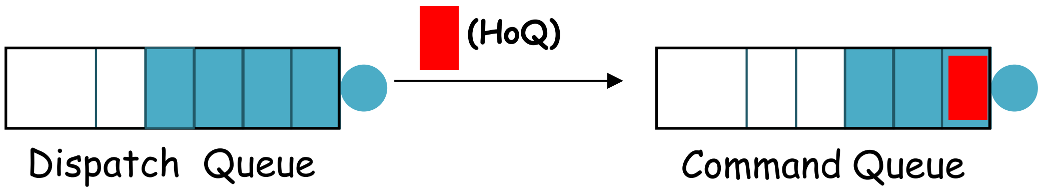
- Avoid out-of-order transfer.
- satisfies $D = X$ without interleaving the requests with DMA transfer!

```
write (A) ;
write (B) ; //set the command priority to 'ORDERED'
```

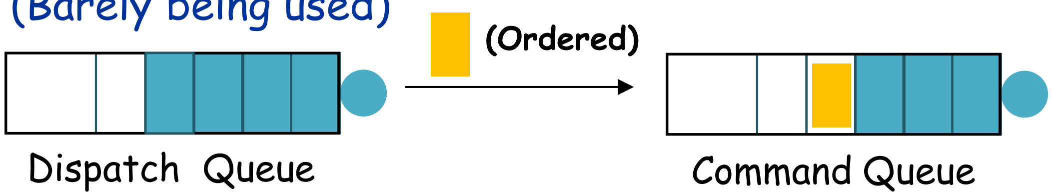


SCSI Command Priority

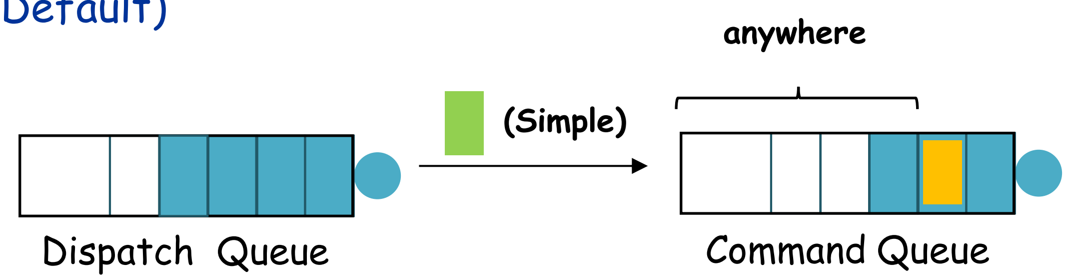
- Head of the Queue



- Ordered (Barely being used)



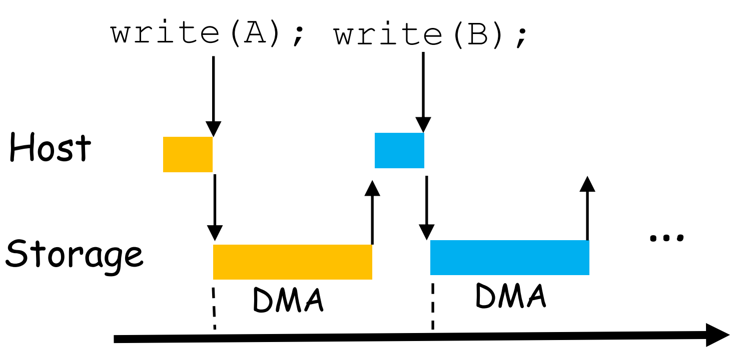
- Simple (Default)



Order Preserving Dispatch

Legacy Dispatch

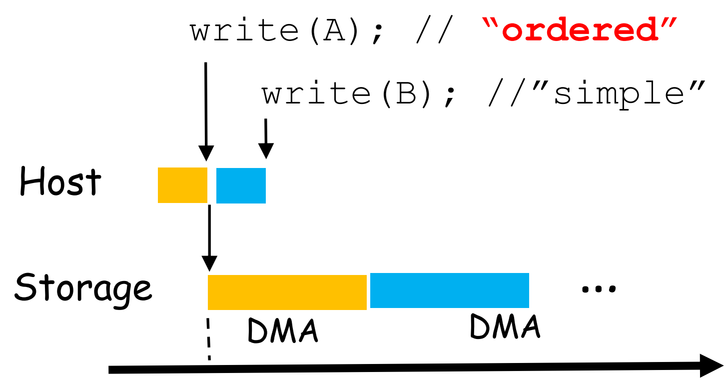
For D = X, wait till DMA finishes to send the following command.



Caller blocks.

DMA transfer overhead

Order Preserving Dispatch



Caller does not block. 👍

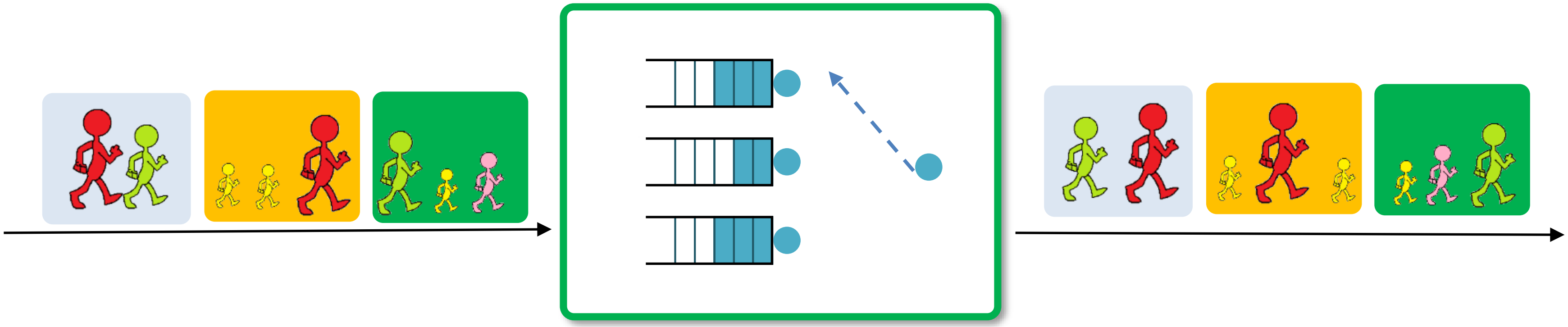
No DMA transfer overhead 👍

With Order Preserving Dispatch, host can control the transfer order without DMA transfer.

$$(I \cancel{=} P) \equiv (I \cancel{=} D) \wedge (D \cancel{=} X) \wedge (X = P)$$

Satisfying I = D

Use NO-OP, or FIFO scheduler.



IO Scheduler

With Epoch Based IO Scheduling, host can control the dispatch order with existing IO scheduler.

$$\cancel{(I = P)} \equiv \cancel{(I = D)} \wedge (D = X) \wedge (X = P)$$

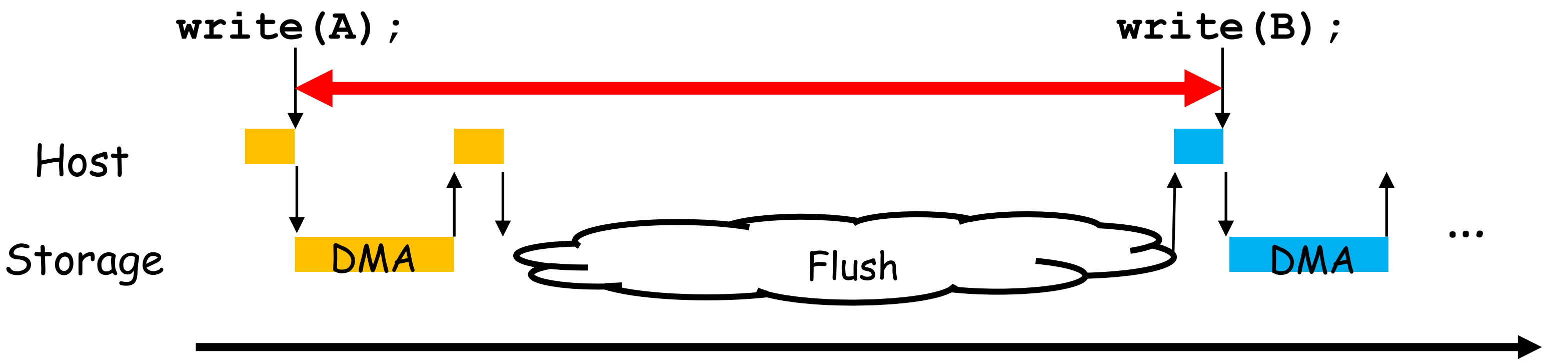
barrier write

Order-preserving dispatch

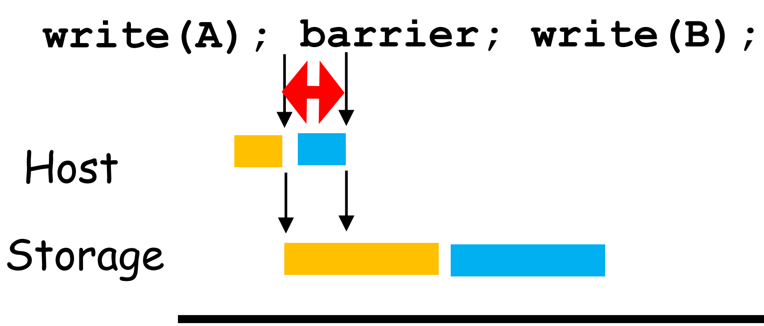
Epoch-based IO scheduler

Enforcing the Storage Order

Legacy Block Layer (With Transfer-and-Flush)



Order Preserving Block Layer

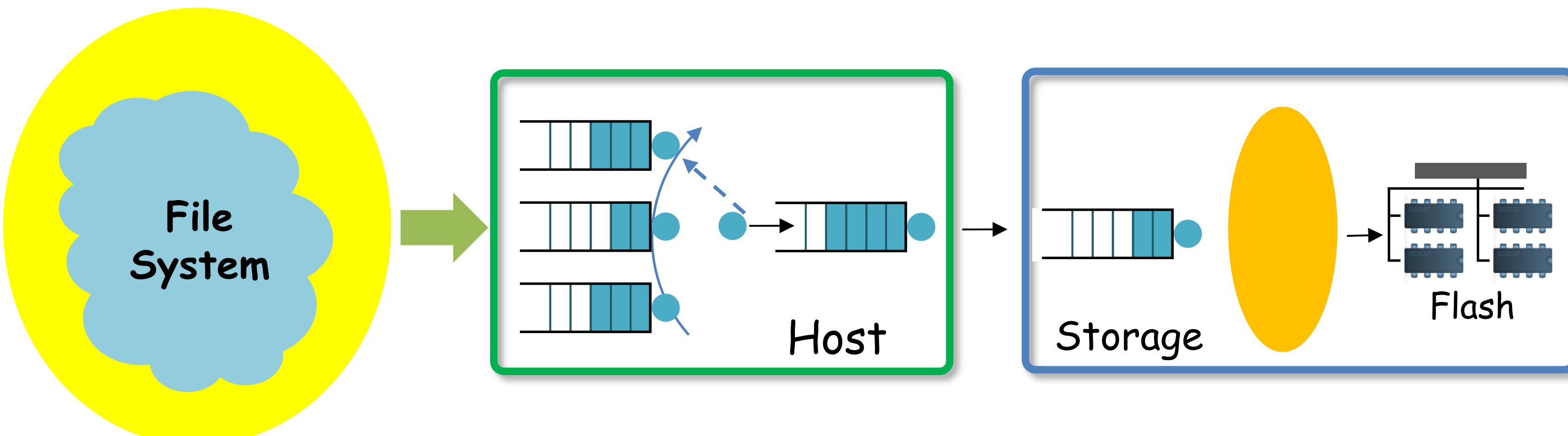


No Flush !

No DMA !

No Context Switch !

fbarrier() and fdatabarrier()

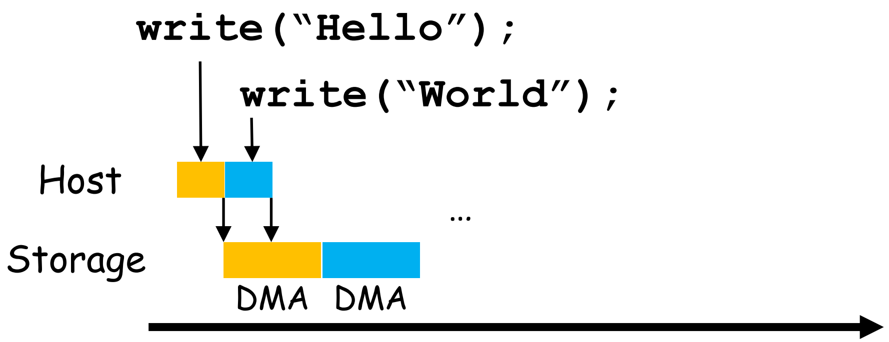


New primitives for ordering guarantee

	Durability guarantee	Ordering guarantee
Journaling	✓ fsync() ➢ Dirty pages ➢ journal transaction ➢ Durable	✓ fbarrier() ➢ Dirty pages ➢ Journal transaction ➢ ~~durable~~
No journaling	✓ fdatasync() ➢ Dirty pages ➢ durable	✓ fdatabarrier() ➢ Dirty pages ➢ ~~durable~~

Separation of Ordering Guarantee and Durability Guarantee

```
write(fileA, "Hello") ;
fdatabarrier (fileA) ;
write(fileA, "World") ;
```



DMA transfer overhead NO

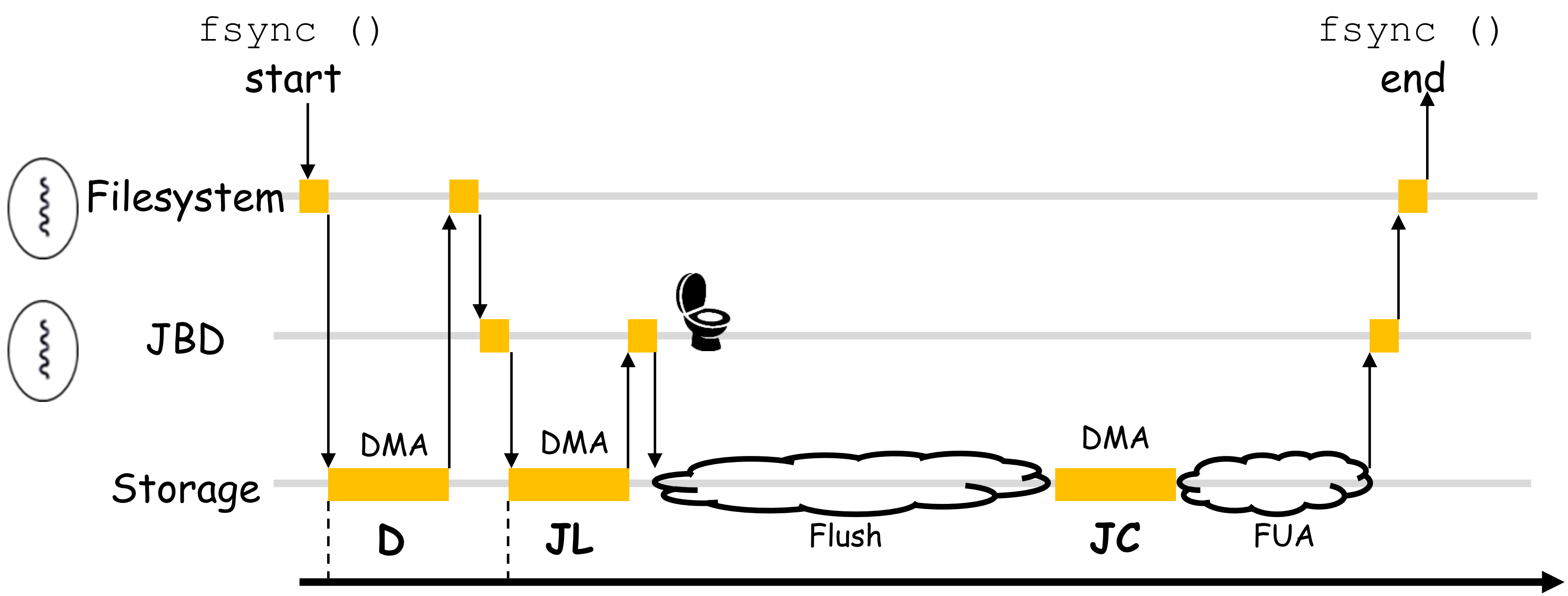
Flush overhead NO

Context switch NO

fsync() in EXT4

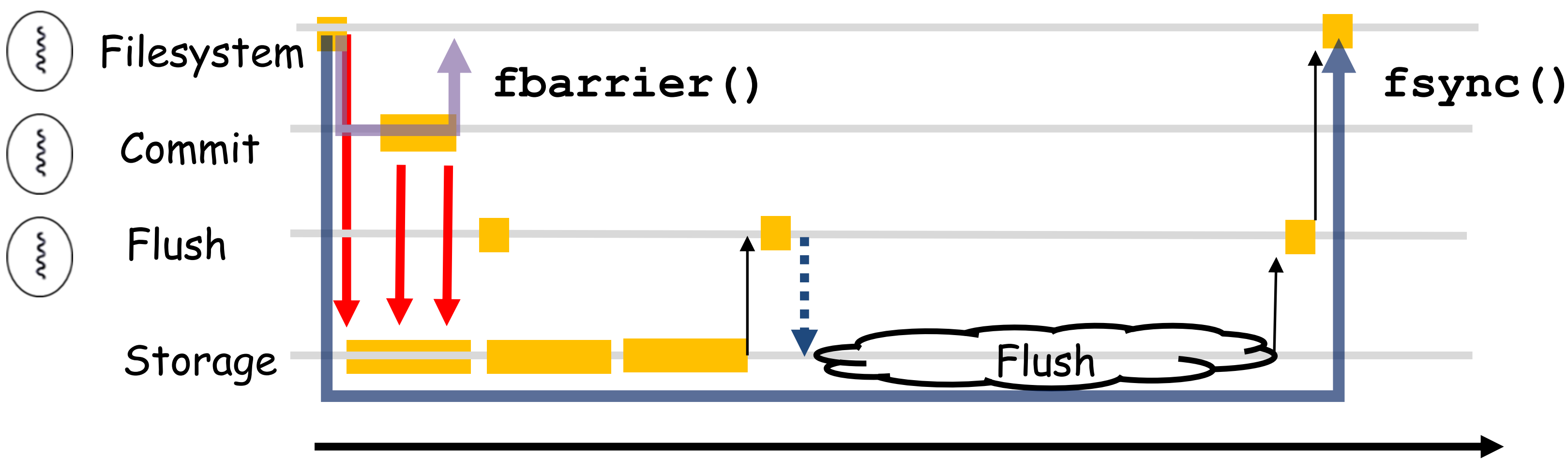
{Dirty Pages (**D**), Journal Logs (**JL**)} → {Journal Commit (**JC**)}

- Two Flushes
- Three DMA Transfers
- A number of Context switches



fsync() and fbarrier() in BarrierFS

- ~~Two~~ One Flushes
- ~~Three DMA Transfers~~
- One ~~A number of~~ Context switch



Outline

1. Background
2. Storage Order in single queue IO stack
3. Storage Order and concurrency
4. Storage Order in multi queue IO stack
5. Conclusion

Concurrency and Order (CJFS, FAST'23)

- What we expected: Concurrent Journaling

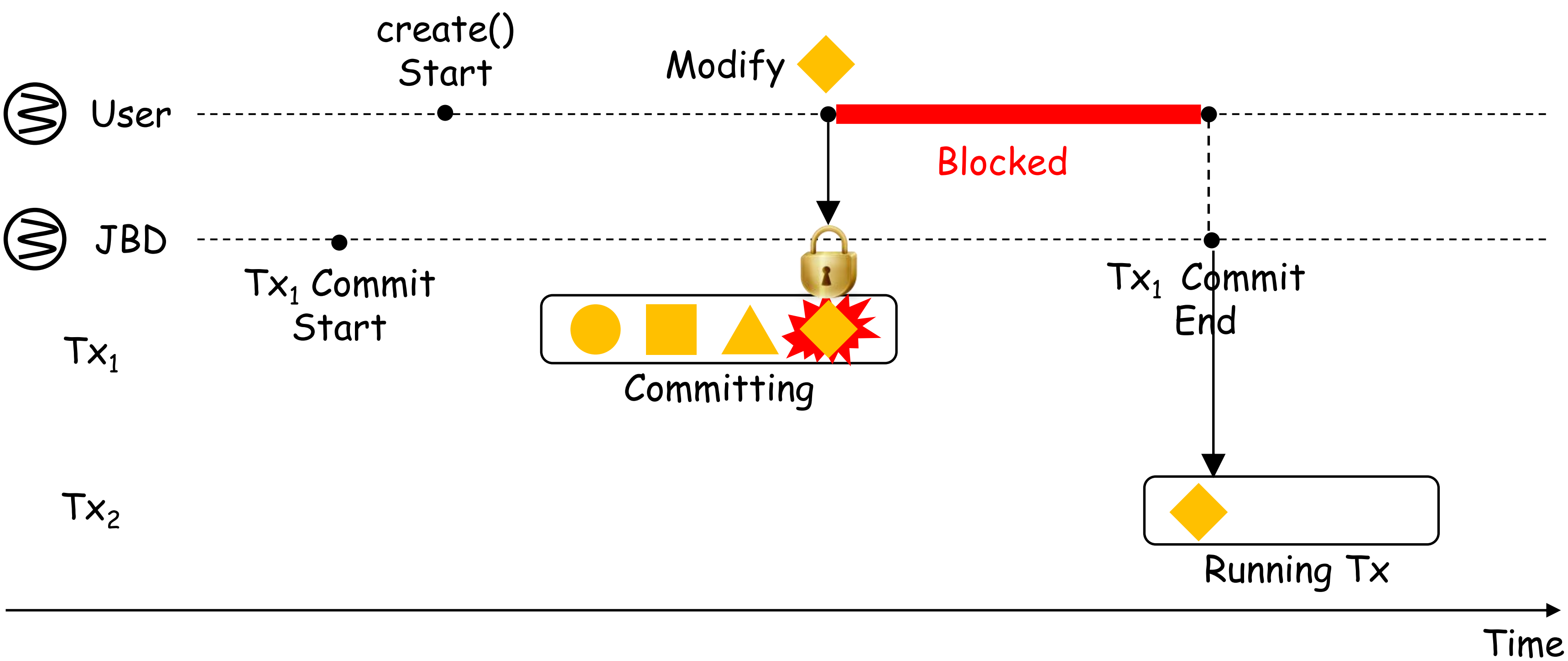
T1 fsync() ......
T2 fsync() ......
T3 fsync() ......
T4 fsync() ......

- What we have observed: Serial Journaling

T1 fsync() ......
T2 fsync() ......
T3 fsync() ......
T4 fsync() ......

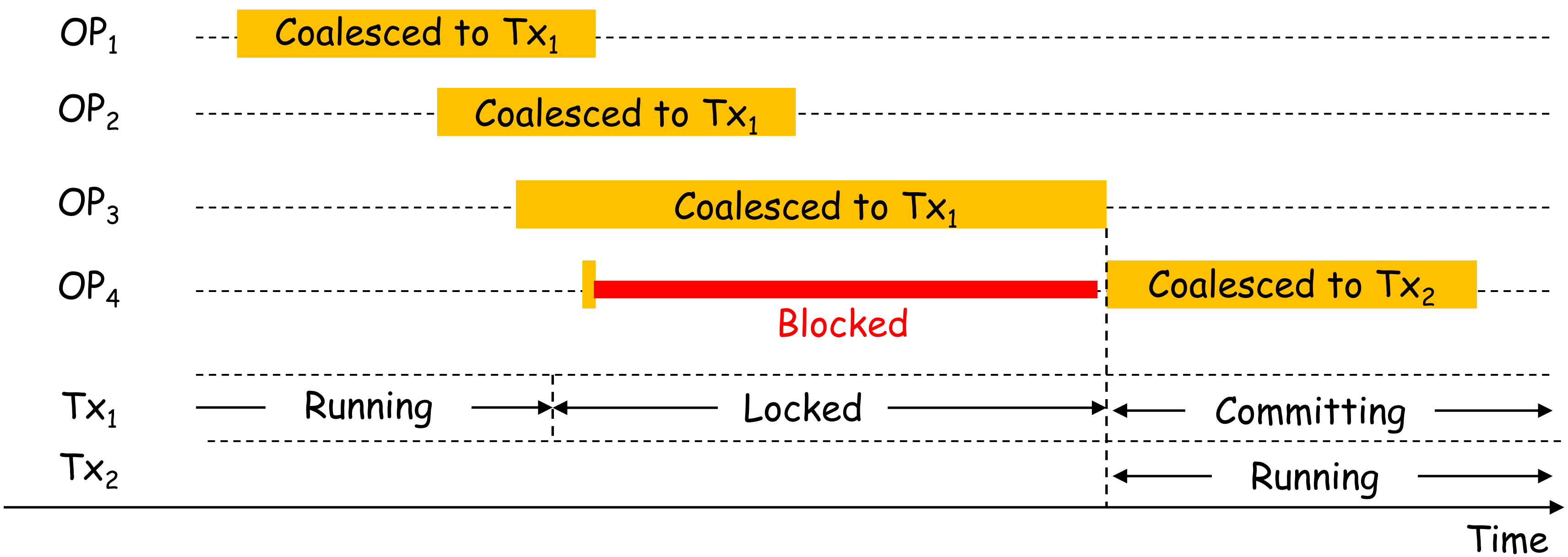
Reason 1: Transaction conflict

- A file operation modifies a page which is being committed.
- A file operation is blocked till the conflict transaction is committed.
- Most journal transactions have some blocks in common; bitmap, superblock

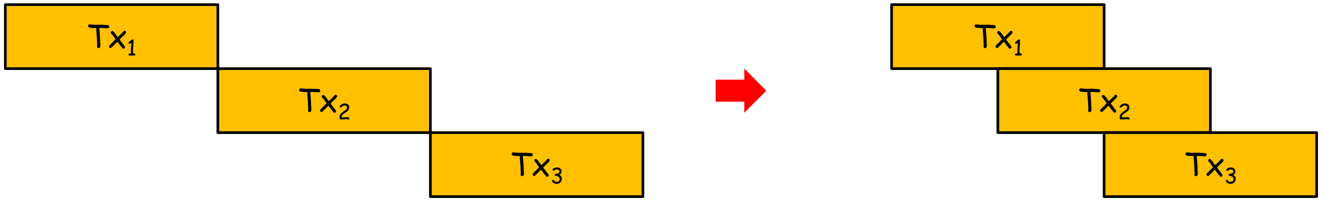


Reason 2: Transaction Lock-Up

- When committing a running transaction, the filesyste stops issues journal handle and waits till all outstanding journal handles are returned.
- During transaction lock-up, a filesystem operation is blocked.

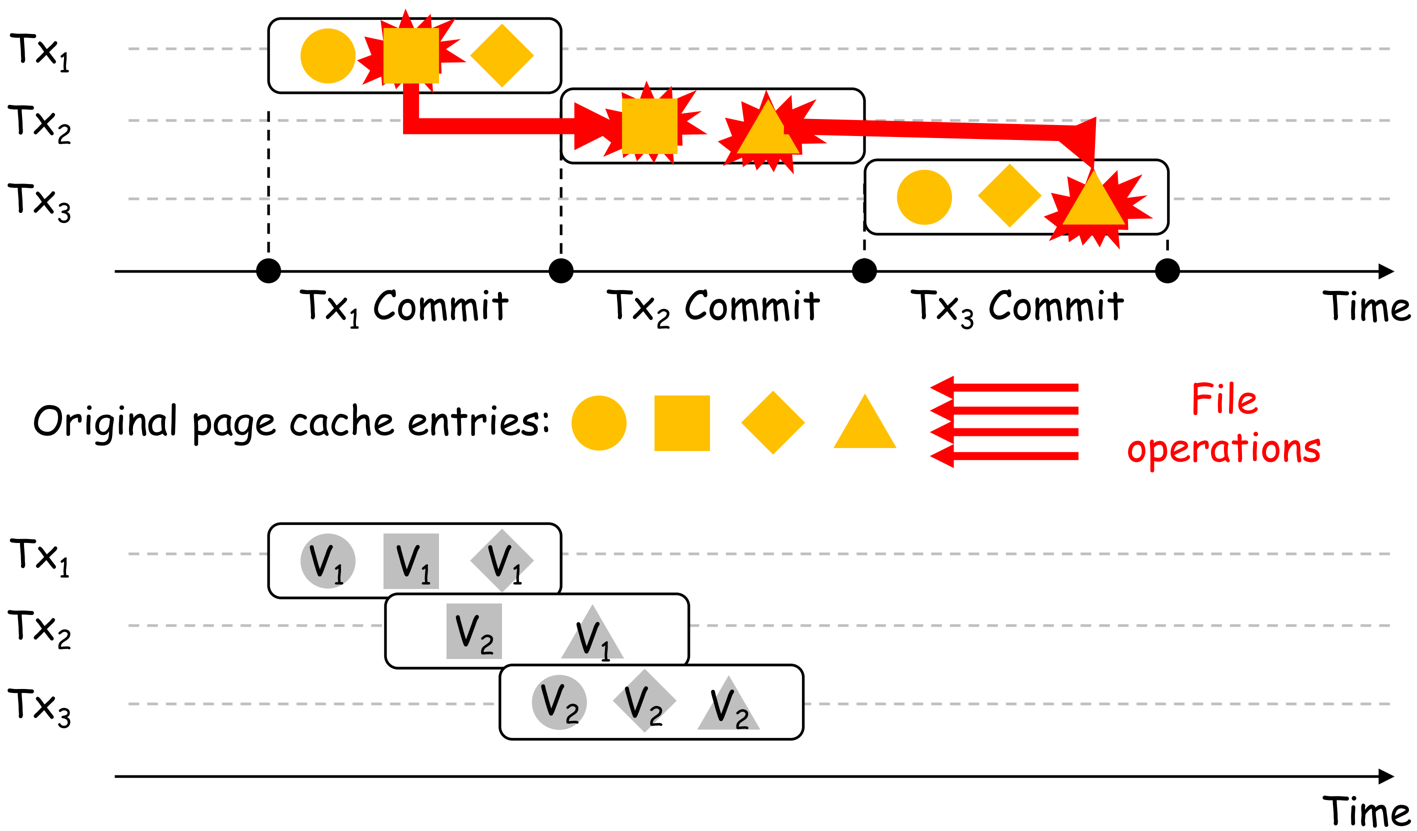


Resolve transaction conflict and transaction lock-up.



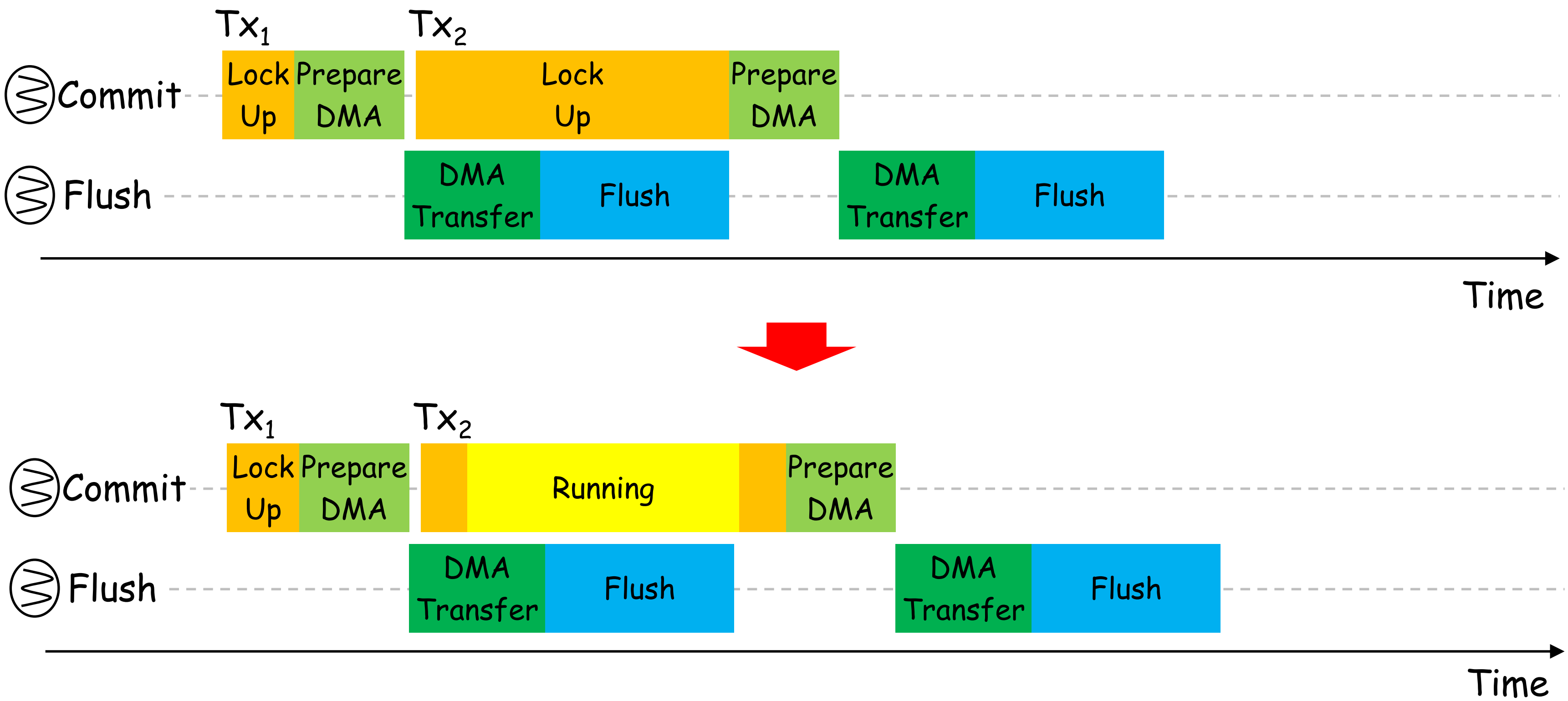
Resolve Transaction Conflict: Multi-version Shadow Paging

- Commit a shadow page rather than the original page.
- Creating a shadow page is not as significant as expected.
- A page can have up to N versions. (currently, N = 5)

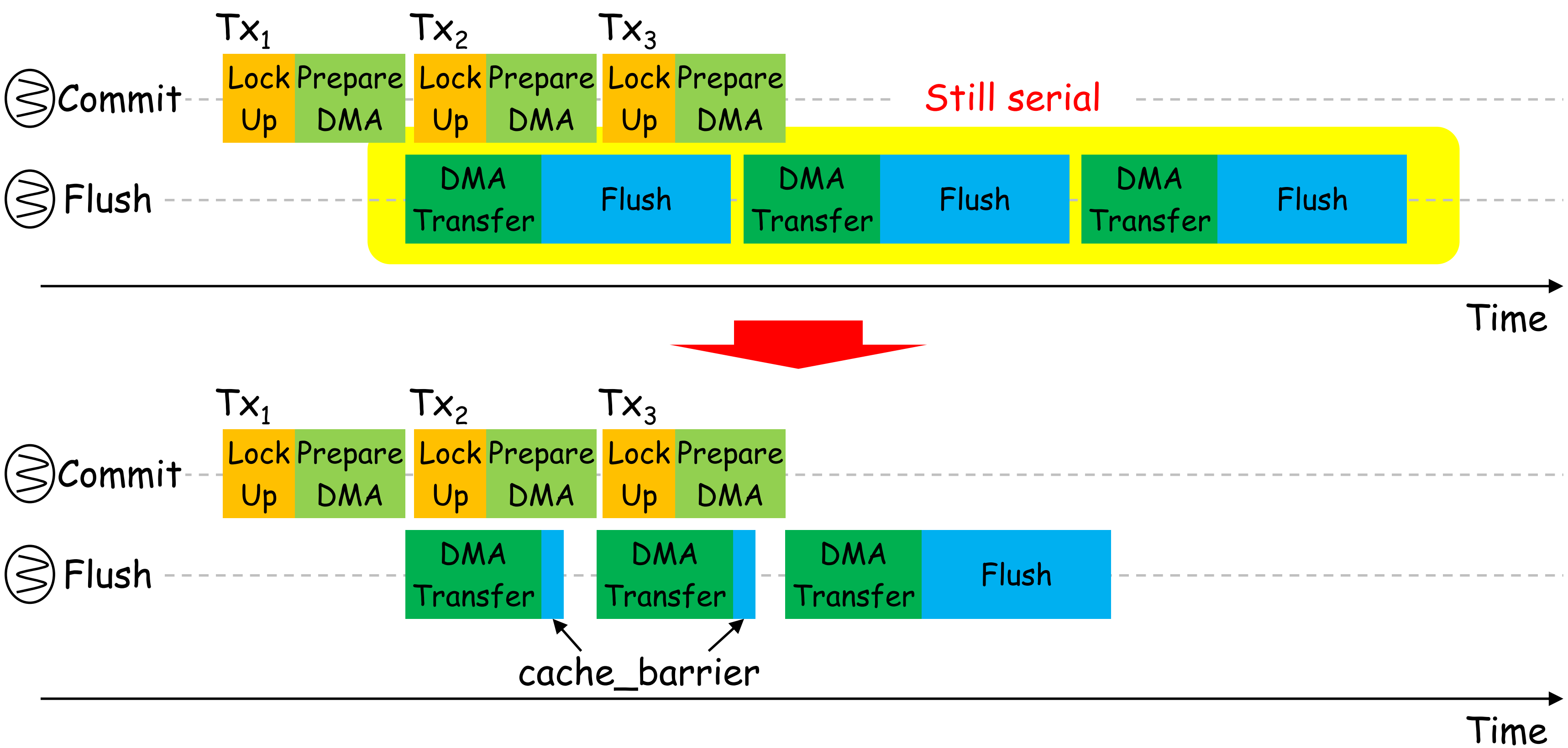


Resolve Transaction Lock-up Overhead: Opportunistic Coalescing

- When versions are exhausted, transaction commits are serialized
- The running transaction is locked and waits for preceding transaction commits



Compound Flush



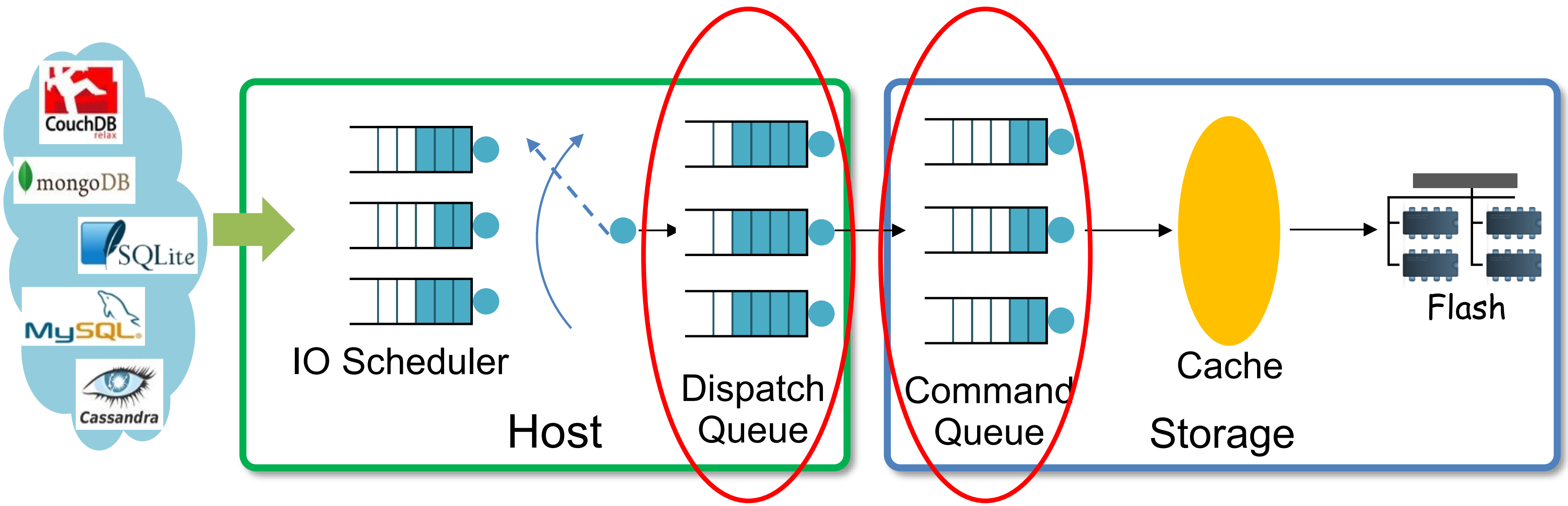
Outline

1. Background
2. Storage Order in single queue IO stack
3. Storage Order and concurrency
4. Storage Order in multi queue IO stack
5. Conclusion

Multi-Queue and Order (OPIMQ, FAST'25)

How can we ensure the order across the queues?

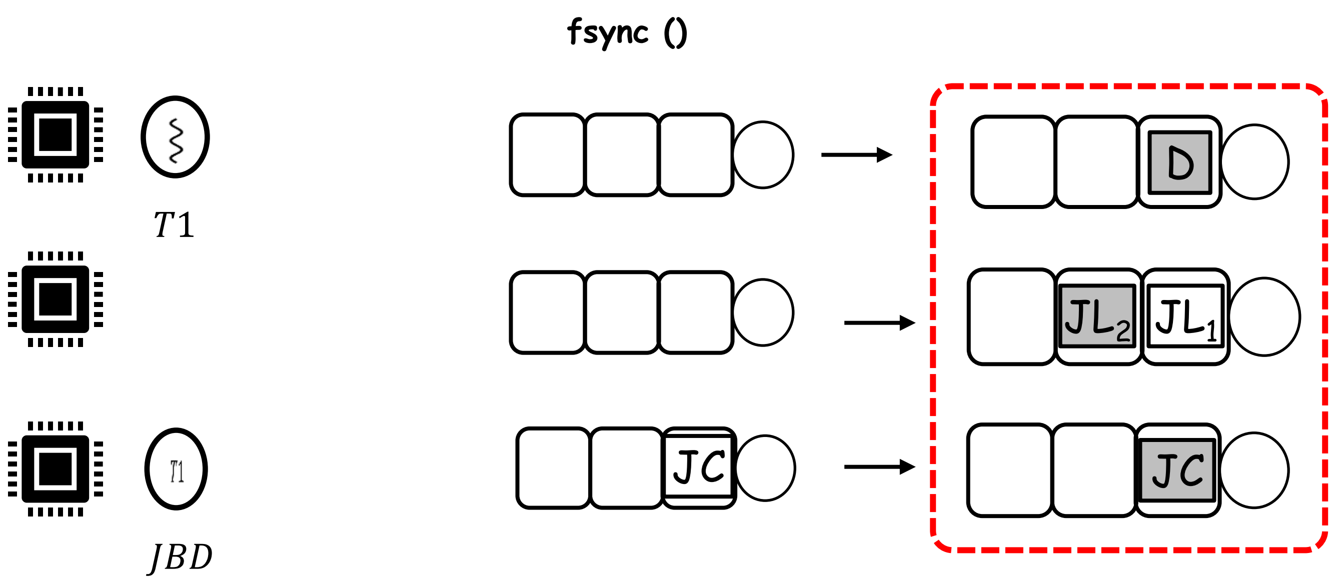
Queues are meant to be independent.



Storage Order in Multi-Queue Block Device

✓ Ensuring the storage order across the **different** queues.

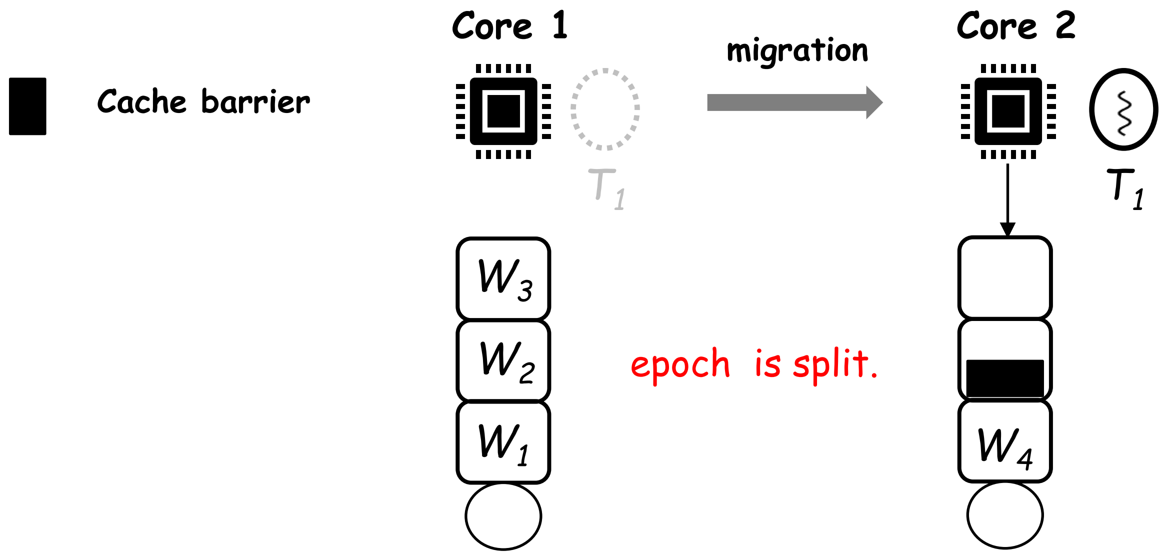
{Dirty Pages (**D**), Journal Logs (**JL**)} $\not\rightarrow$ {Journal Commit (**JC**)}



Inter-Queue Storage Order Dependency

✓ When requests are from same thread

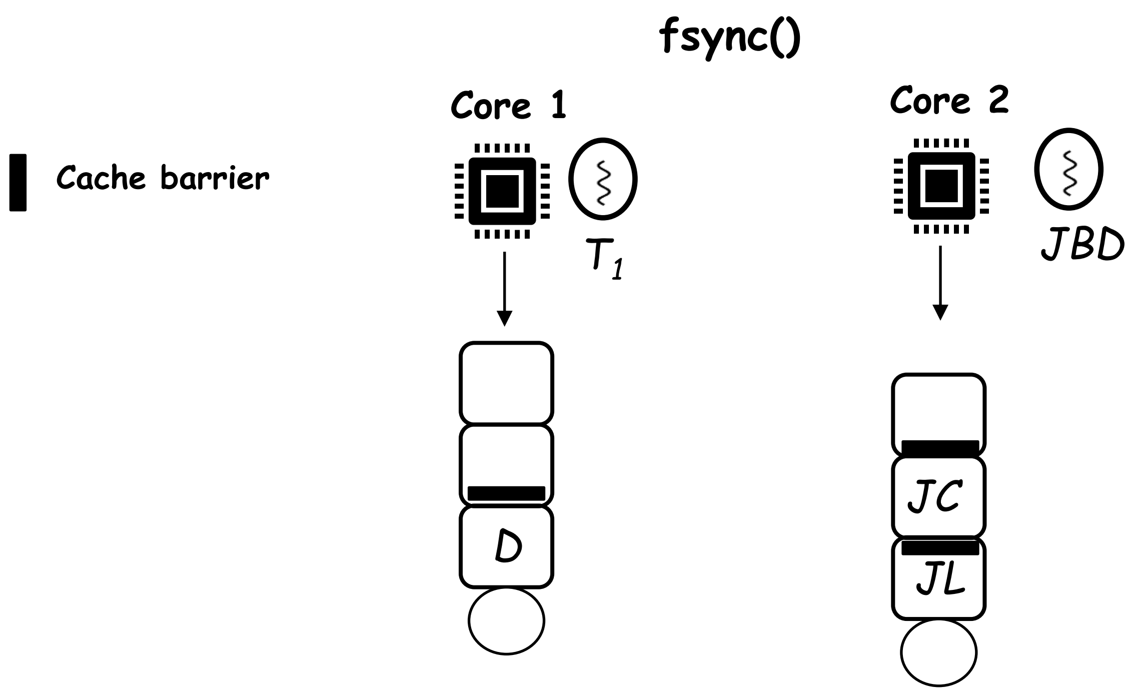
- What we want: {W1,W2,W3,W4} →{…}
- What may happen: {W4} → {W1,W2,W3,…}



Inter-Queue Storage Order Dependency

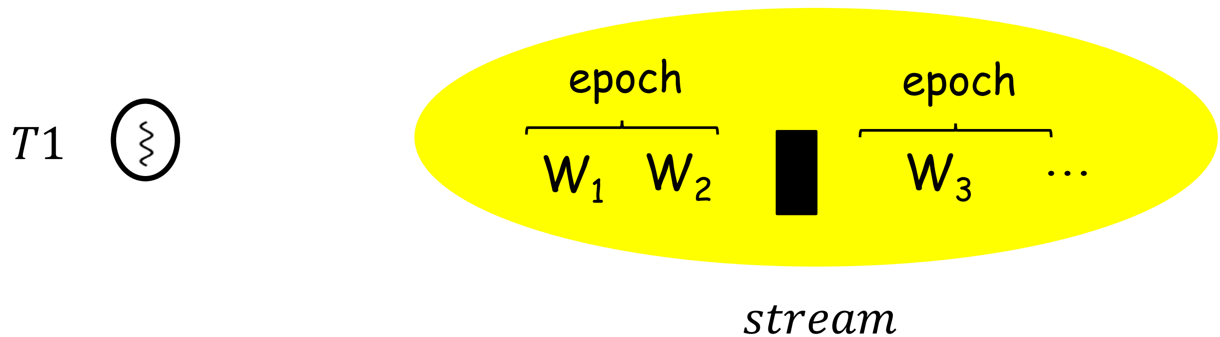
✓ When requests are from different threads.

- What we want: {D}→{JL}→ {JC}
- What may happen: {JL} →{JC} →{D}



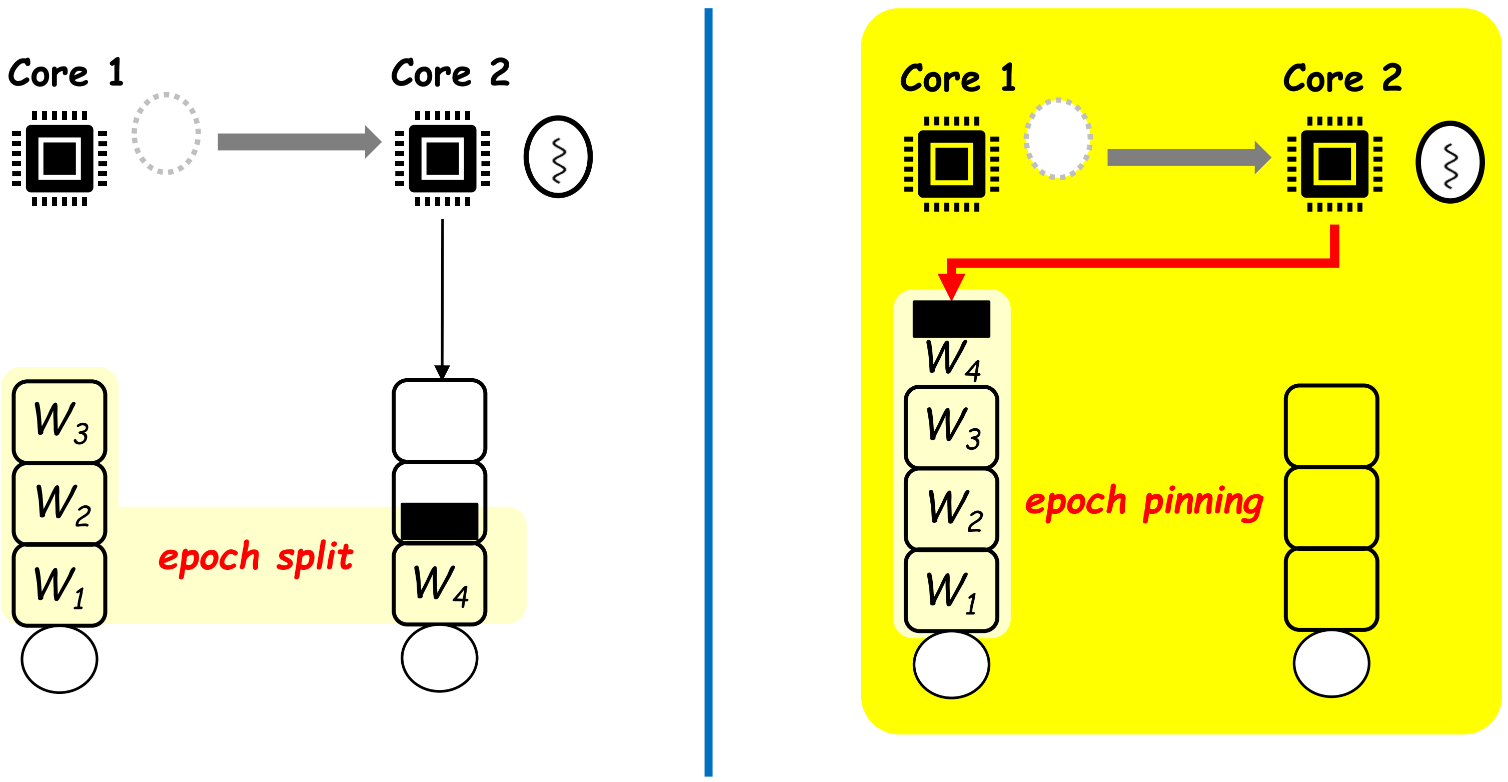
Model

- Stream: a set of IO requests generated by the same thread.
- Epoch
  - A set of order-preserving write requests that can be reordered or coalesced with each other
  - Cache barrier command delimits the boundaries of an epoch.
- Write command has <stream id, epoch id>

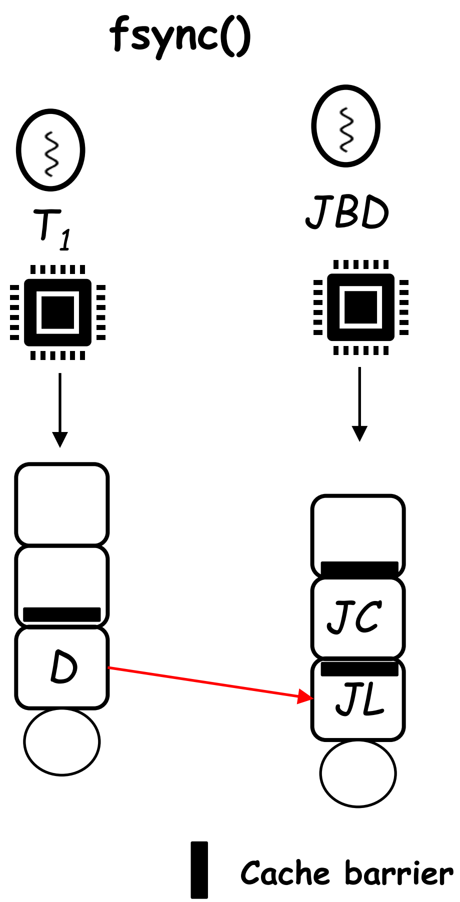


Inter-queue order within a thread: Epoch Pinning

IO's in the same epoch are placed at the same queue.



Inter-queue order among the threads: Dual Stream Write



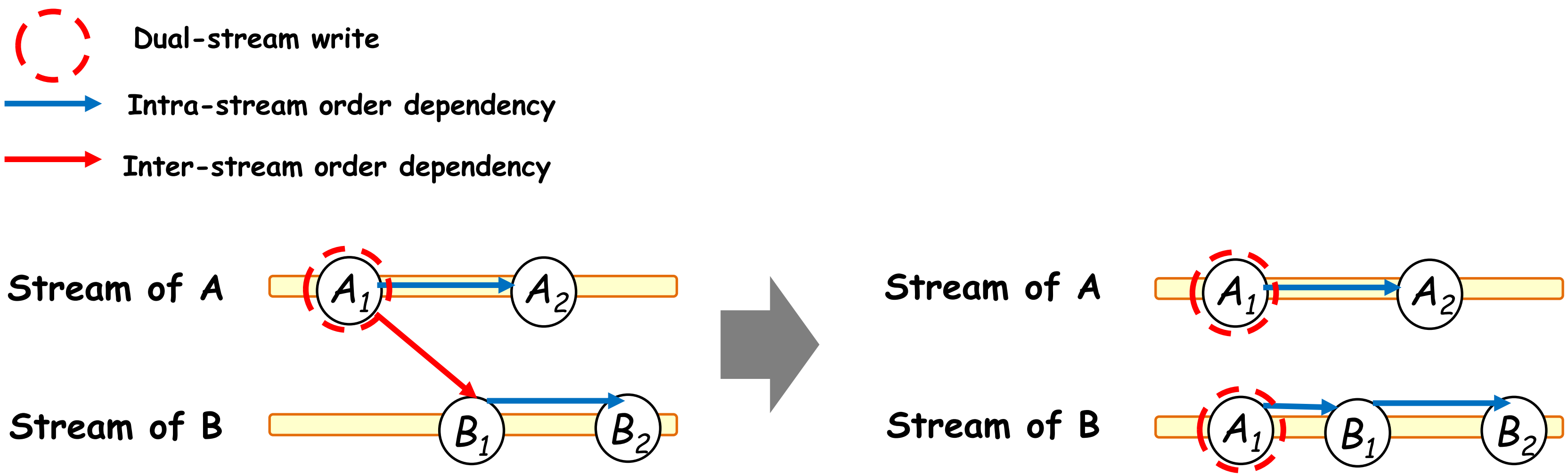
Dual Stream Write

A write request that belongs to two streams.

major <stream id, epoch id> and

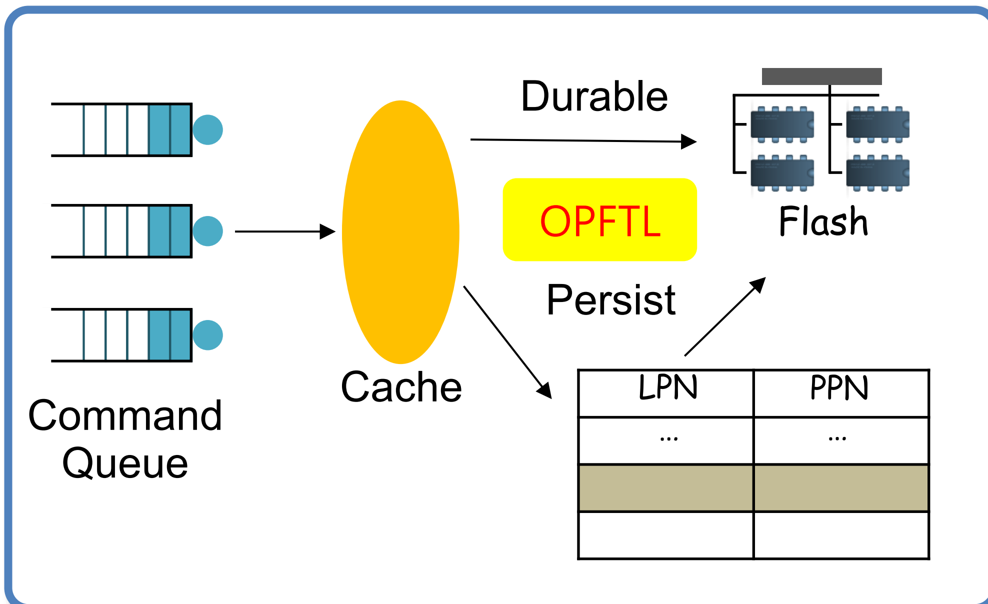
minor <stream id, epoch id>

Dual Stream Write



Order-Preserving FTL

- ✓ Order-preserving mapping table update.
- ✓ Epochs are made persisted in order within a stream.
- ✓ For the dual-stream write, guarantee the persistence order in both streams.



Storage

Outline

1. Background
2. Storage Order in single queue IO stack
3. Storage Order and concurrency
4. Storage Order in multi queue IO stack
5. Conclusion

Conclusion

- ✓ Why transfer-and-flush?
  - Host does not trust storage.
  - Host needs to ensure the every step, e.g. data transfer, FLUSH.
  - "cache barrier"? Standardize in UFS, but in NVMe is still pending.
  - OS needs to run correctly on thousands of different and possibly unreliable storage models.
- ✓ Why core migration causes ordering issue?
  - OS design is CPU centric and command Queue is bound to CPU.
  - Thread is migrated to new CPU, IO command is fed to new queue.

Question?