Barrier Enabled IO Stack for Flash Storage

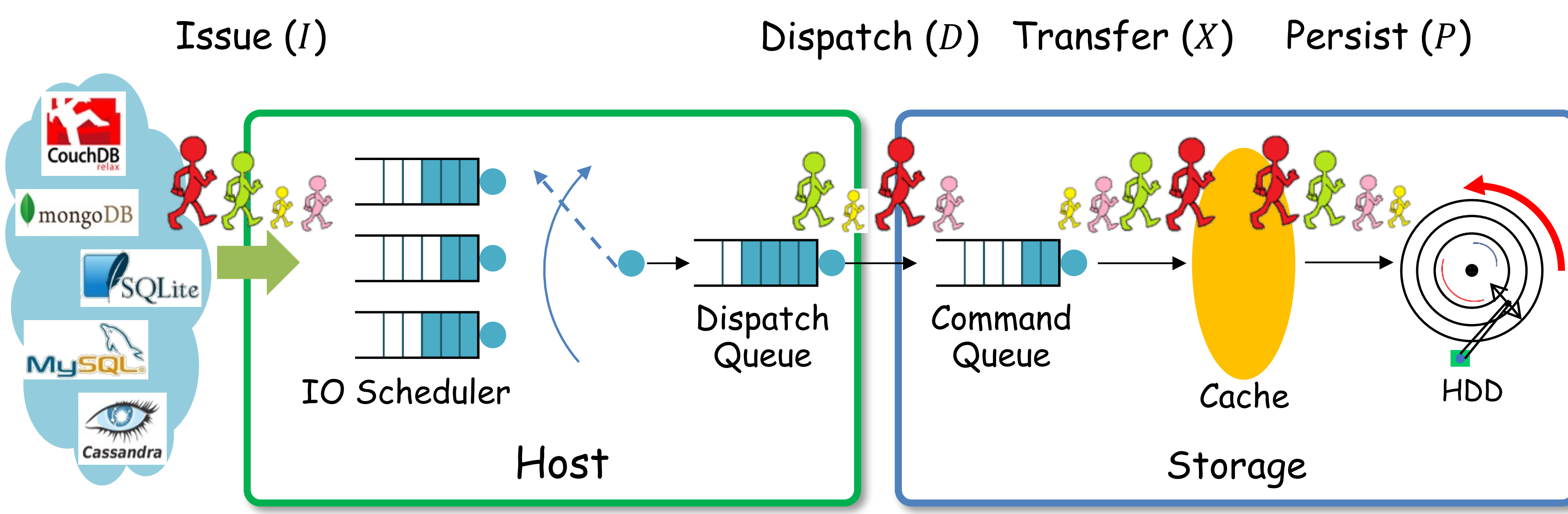
Youjip Won, Jaemin Jung, Gyeongyeol Choi,
Joontaek Oh, Seongbae Son, Jooyoung Hwang, Sangyeun Cho

Hanyang University
Texas A&M University
Samsung Electronics

Motivation

Modern IO Stack

Modern IO stack is Orderless.



$I \neq D$: IO Scheduling

$D \neq X$: Time out, retry, command priority

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

Storage Order

Storage Order: The order in which the data blocks are made durable.

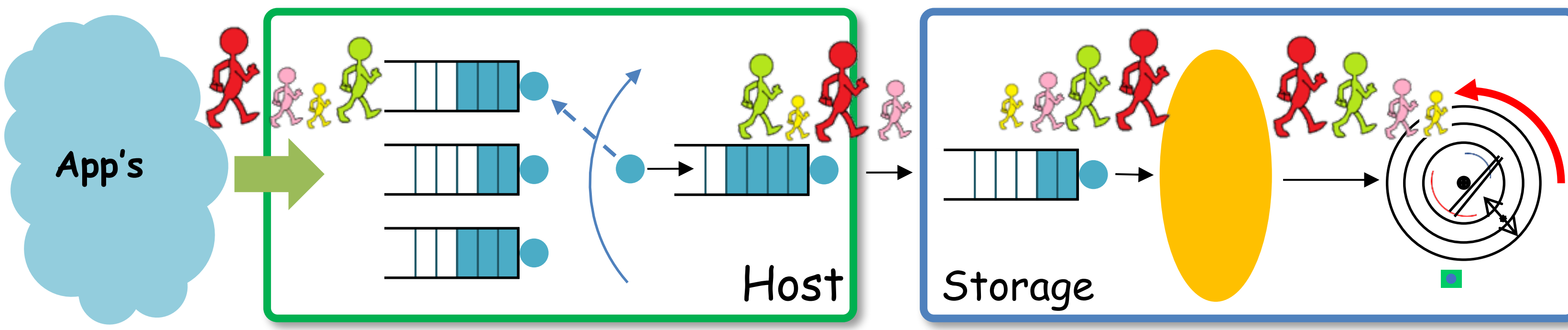
Guaranteeing the storage order

Storage order guarantee

Issue (I) ⟷ Persist (P)

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

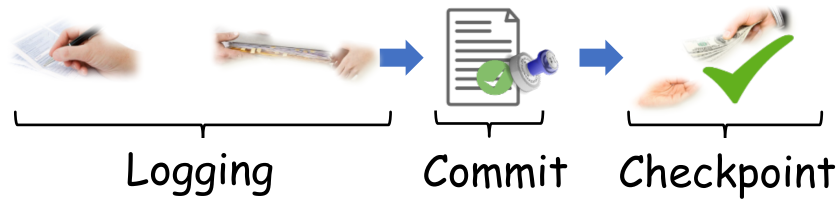
Issue (I) ⟷ Dispatch (D) ⟷ Transfer (X) ⟷ Persist (P)

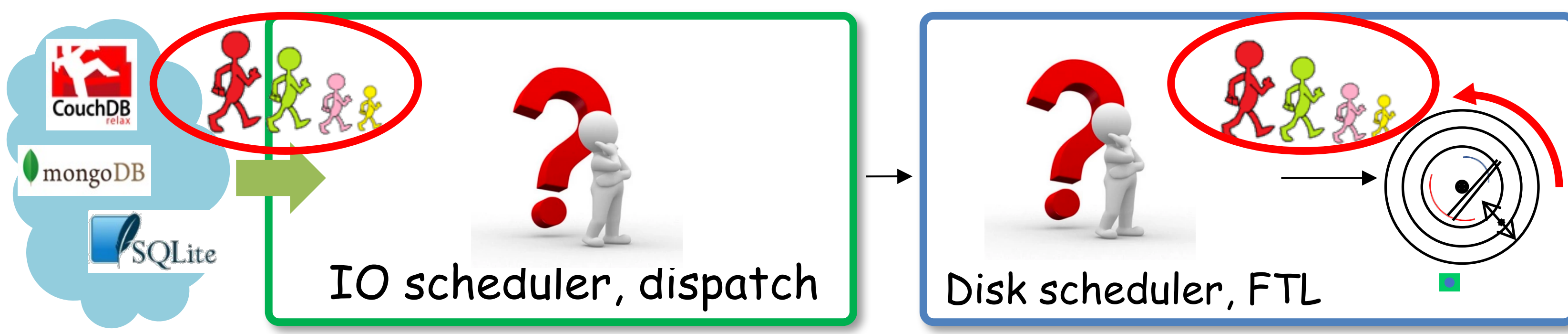


Controlling the Storage Order

Applications need to control the storage order.

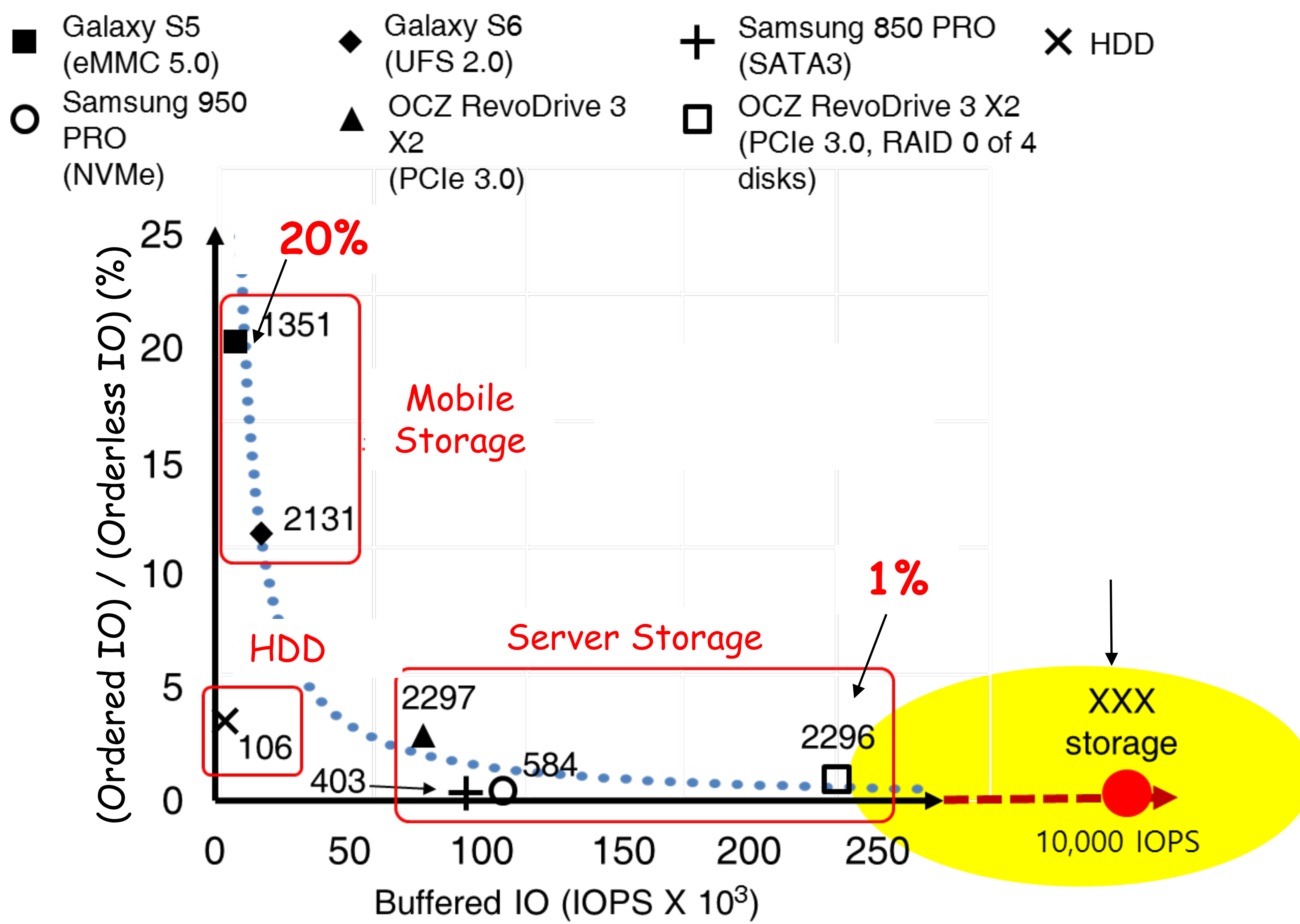
- Database logging
- Filesystem Journaling
- Soft-updates
- COW based filesystem





What's Happening Now....

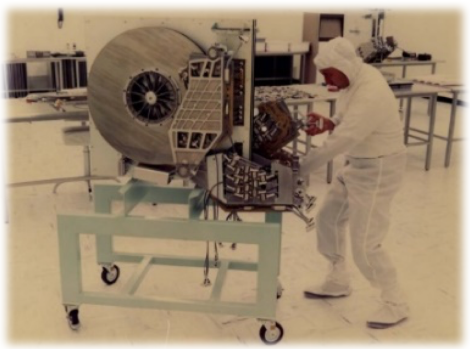
Overhead of storage order guarantee: write() + fdatasync()



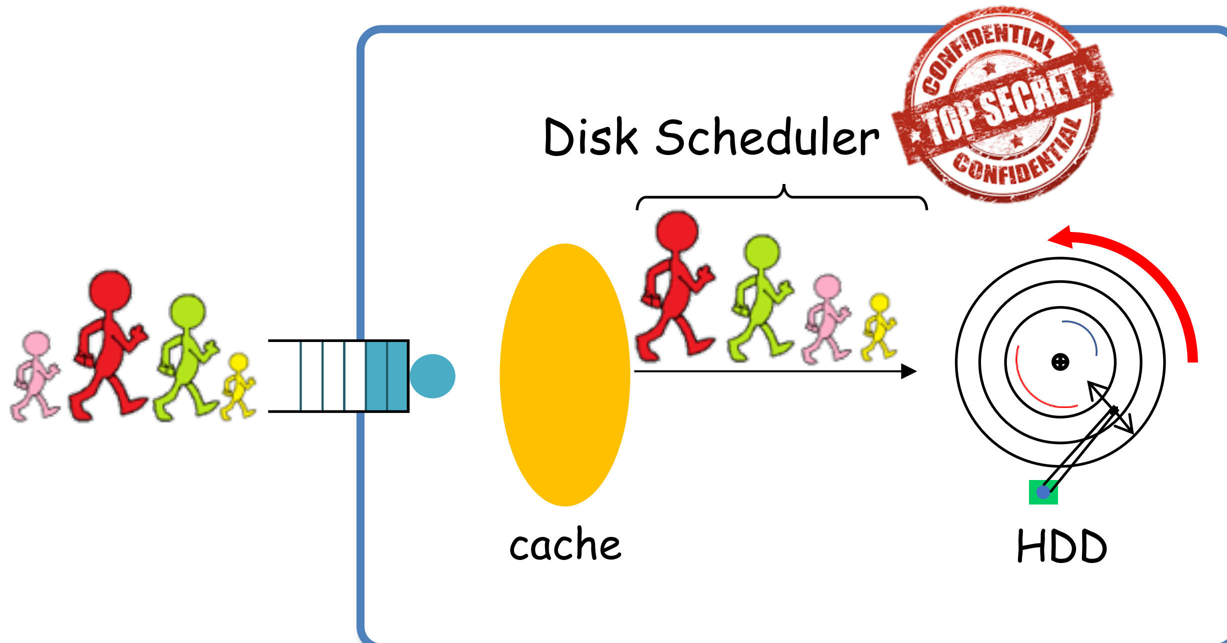
Why has IO stack been orderless for the last 50 years?

In HDD, host cannot control the persist order.

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



250MB @ 1970's



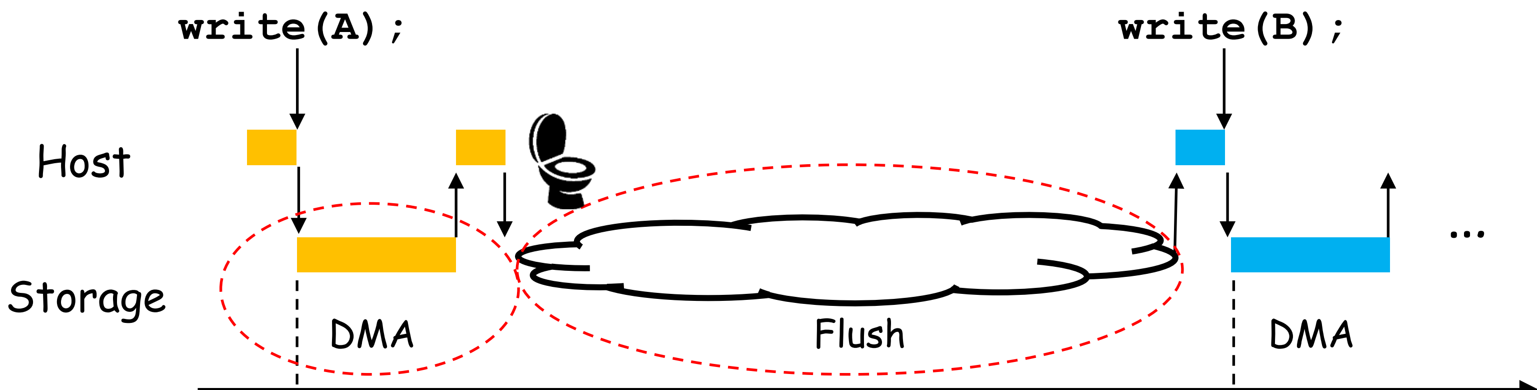
Enforcing Storage Order in spite of Orderless IO Stack

Interleave the write request with Transfer-and-Flush

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

→

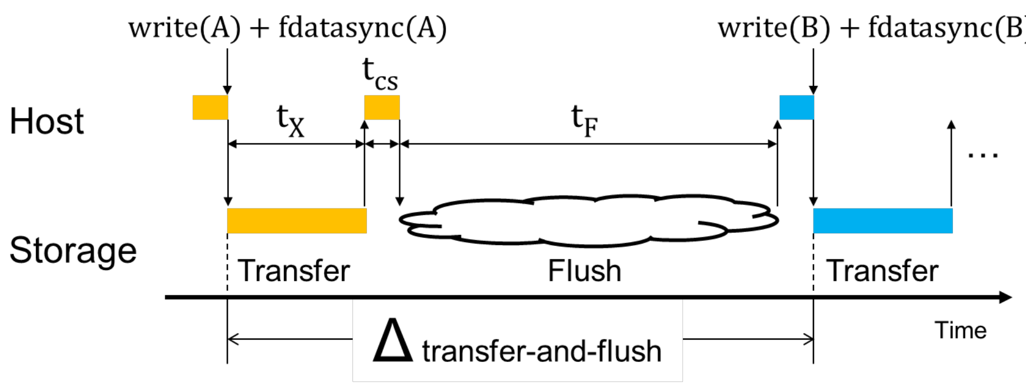
```
write (A) ;
Transfer-and-flush;
write (B) ;
```

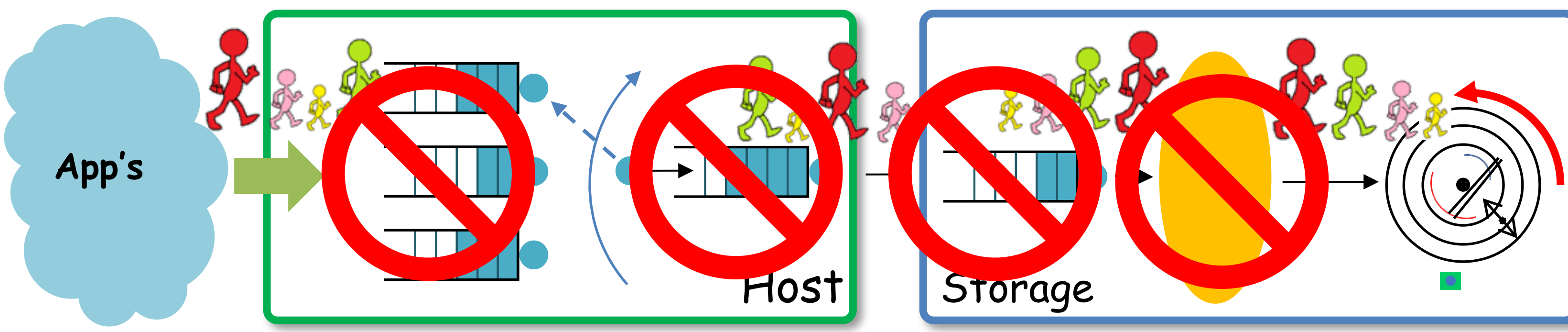


To enforce transfer order, block the caller !

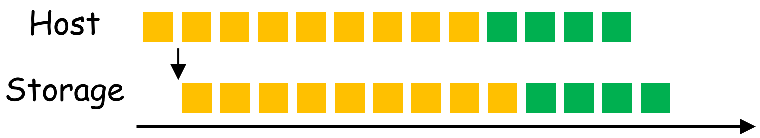
To enforce persist order, drain the cache !

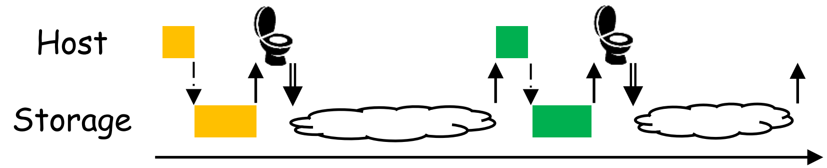
Transfer-and-Flush





Overhead of Transfer-and-Flush



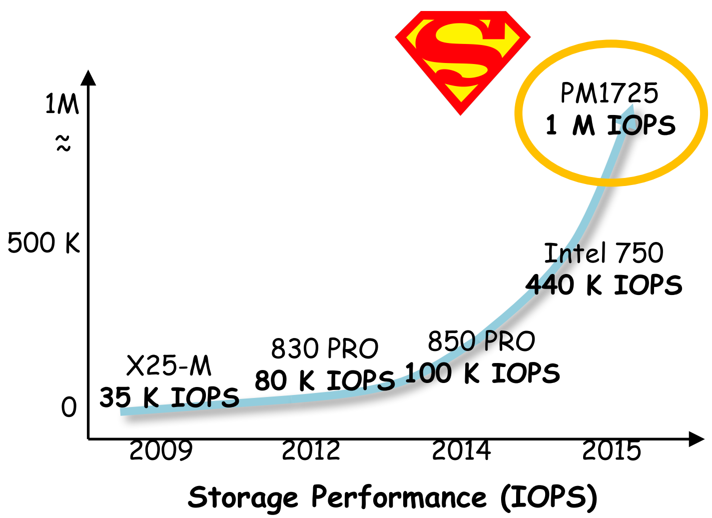


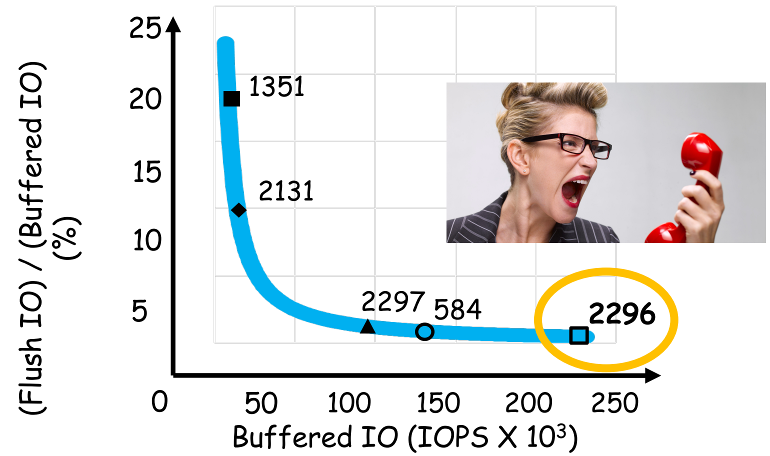
NVMe PM1725
120K IOPS

Ordering Guarantee

< 2%

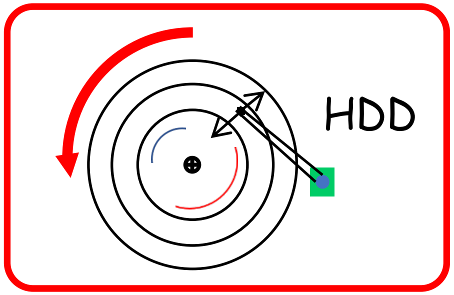
NVMe PM1725
2K IOPS





Developing Barrier-enabled IO Stack

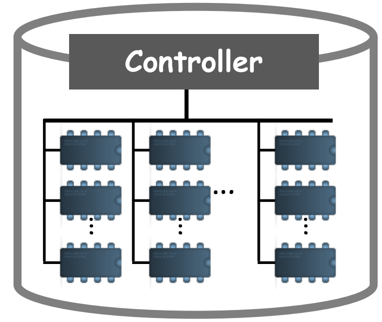
In the era of HDD
(circa 1970)



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)

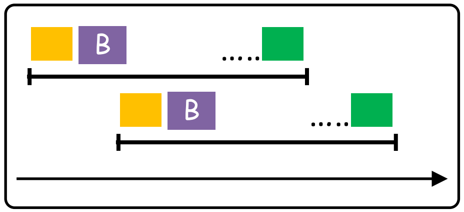


~~Seek and rotational delay~~

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

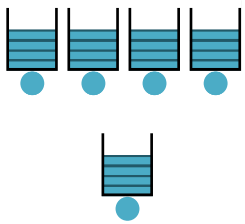
It is a time to re-think the way to control the storage order.

Barrier-enabled IO Stack



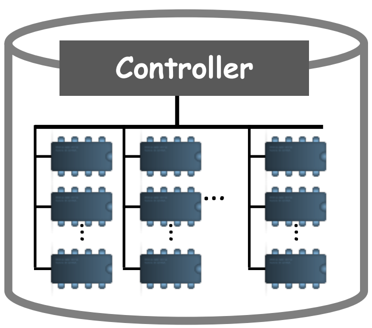
BarrierFS

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



Order-preserving Block Device Layer

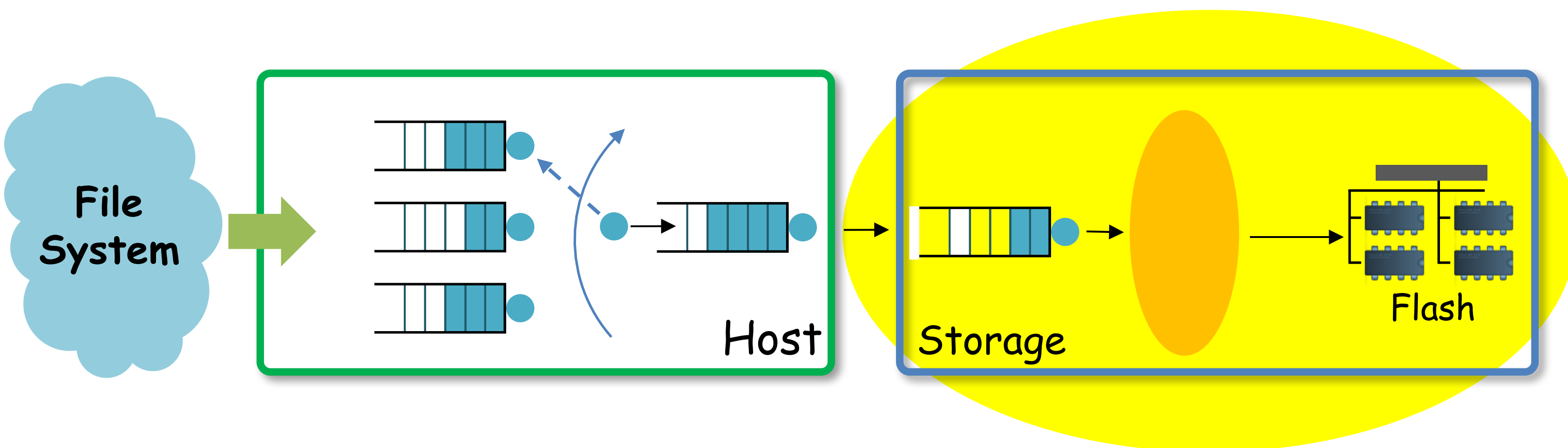
- Order-preserving dispatch
- Epoch-based IO scheduling



Barrier-enabled Storage

- Barrier write command

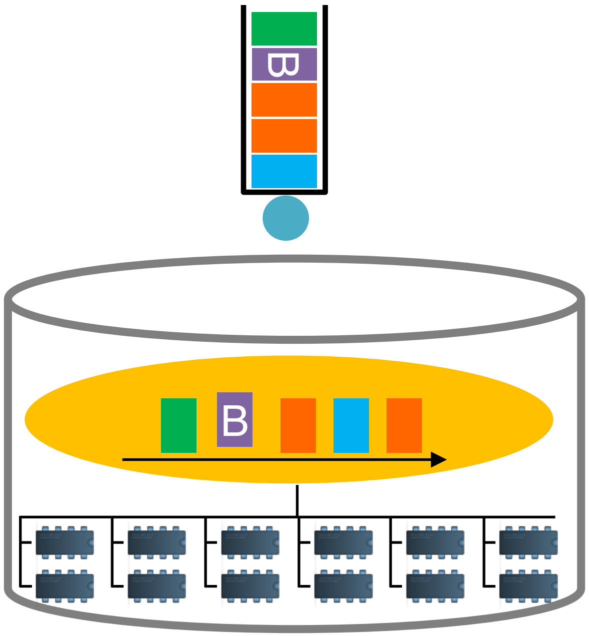
Barrier-enabled Storage



To Control the Persist Order, X = P

barrier command (2005, eMMC)

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



Barrier Write

```
write ;
barrier ;
```

single command →

```
barrier-write ;
```

```
write   A
write   B
barrier-write C
-----------------
write   D
```

Persisted before

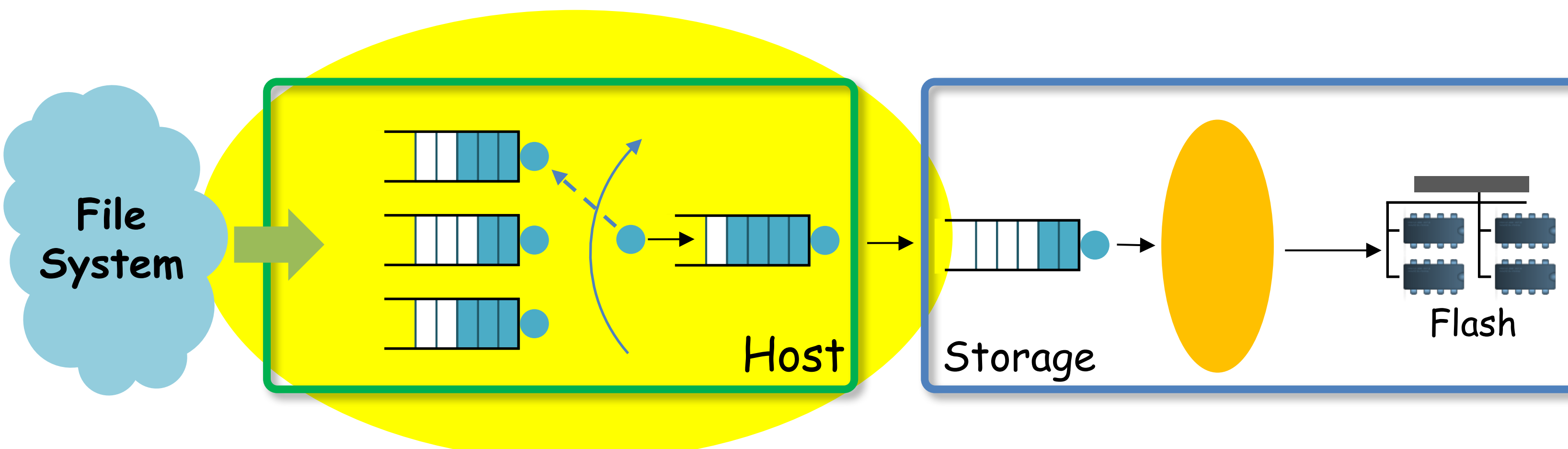
Persisted after

With Barrier Write command,

host can control the persist order without flush.

$$(I \cancel{\preceq} P) \equiv (I \cancel{\preceq} D) \land (D \cancel{\preceq} X) \land (X \cancel{\preceq} P)$$

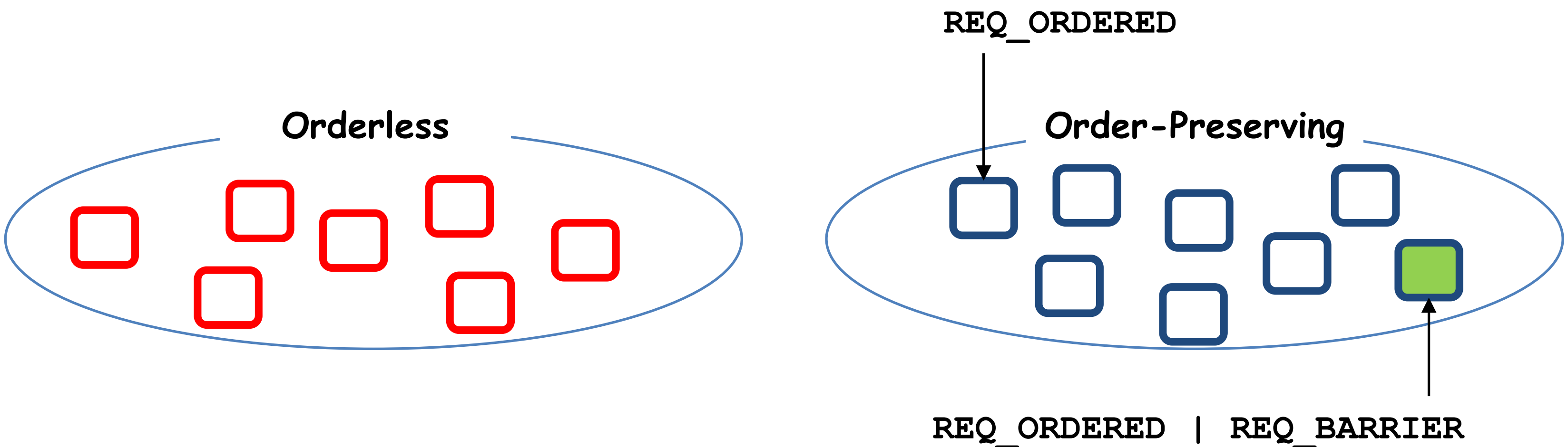
Order-preserving Block Device Layer

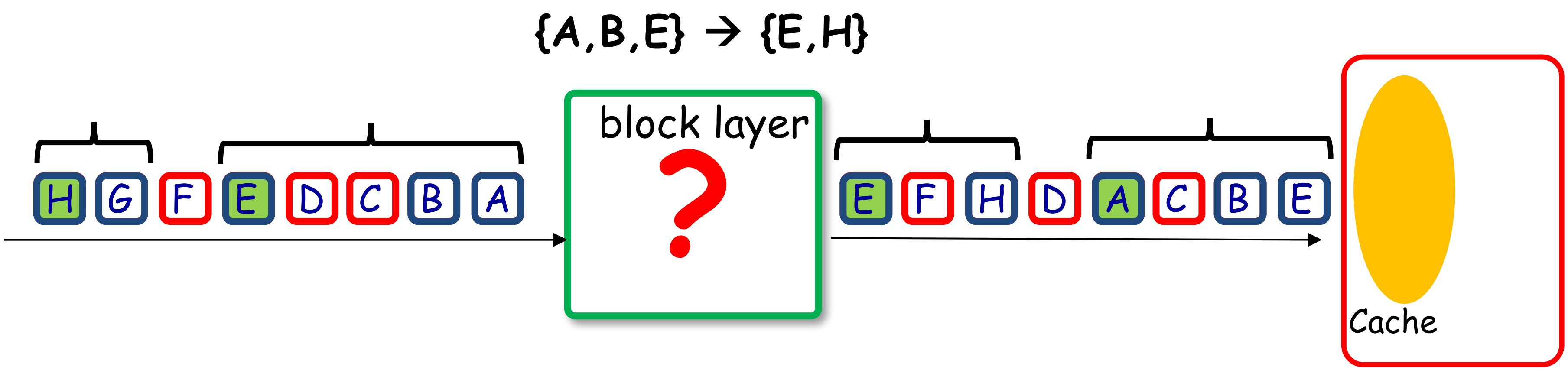


Order Preserving Block Device Layer

- ✓ New request types
- ✓ Order Preserving Dispatch
- ✓ Epoch Based IO scheduling

Request Types



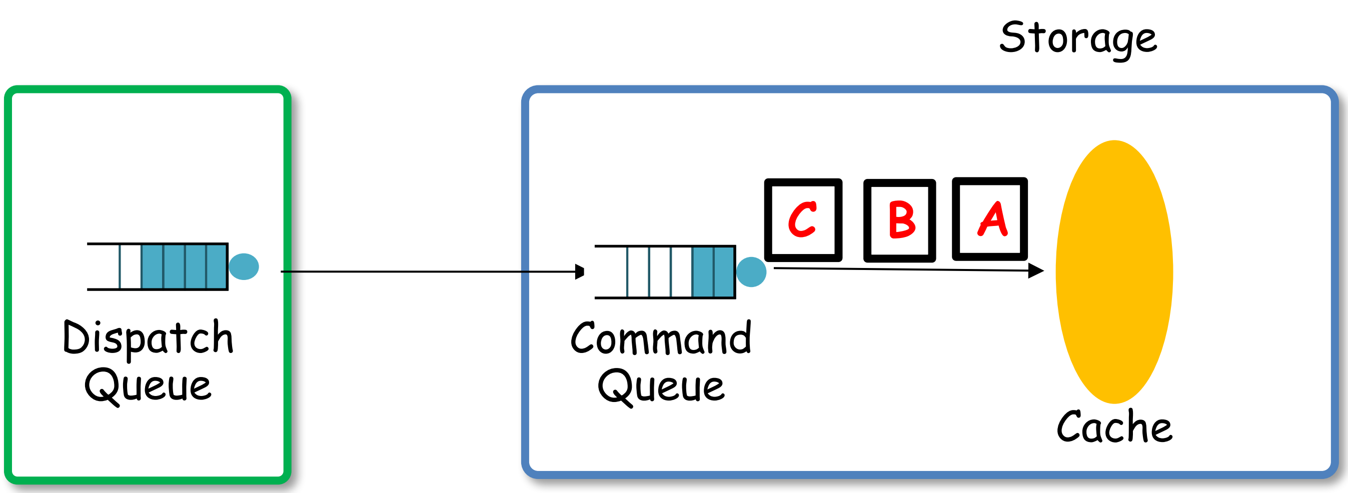


Order Preserving Dispatch Module (for D = X)

- Ensure that the barrier request is serviced in-order.

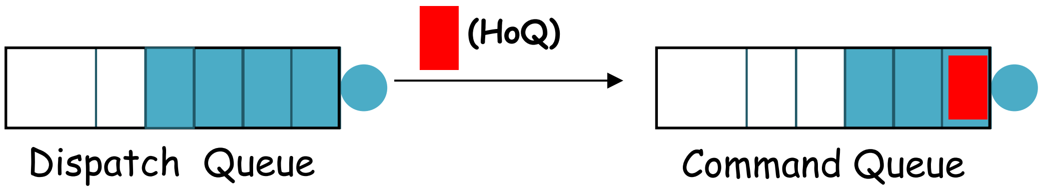
Set the command priority of 'barrier' type request to ORDERED.

```
write A
barrier-write B //set the command priority to 'ORDERED'
write C
```

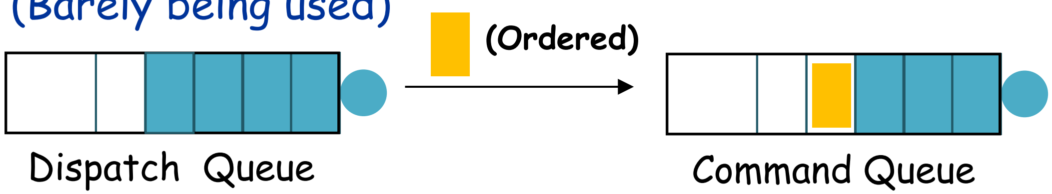


SCSI Command Priority

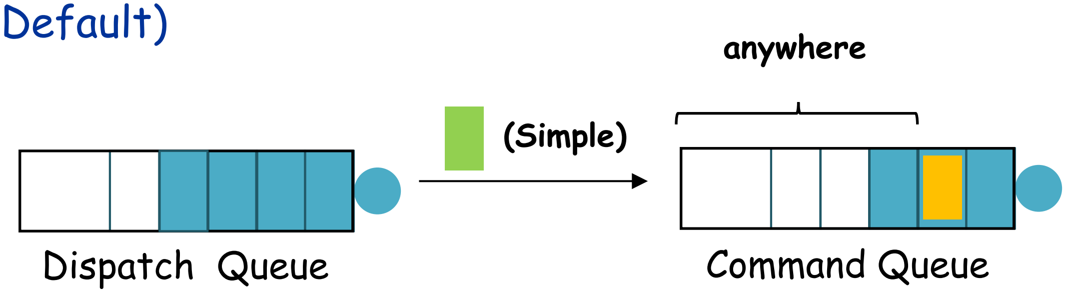
- Head of the Queue



- Ordered (Barely being used)

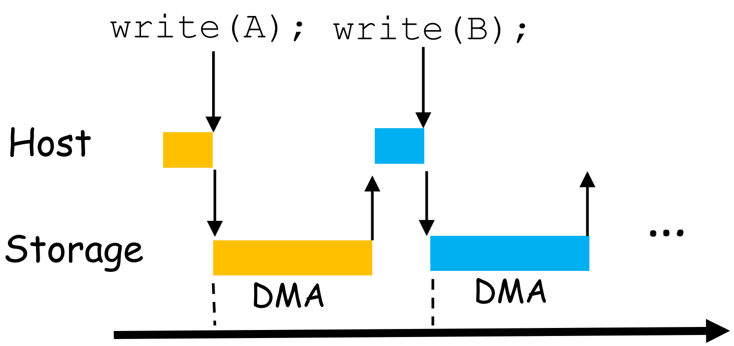


- Simple (Default)



Order Preserving Dispatch

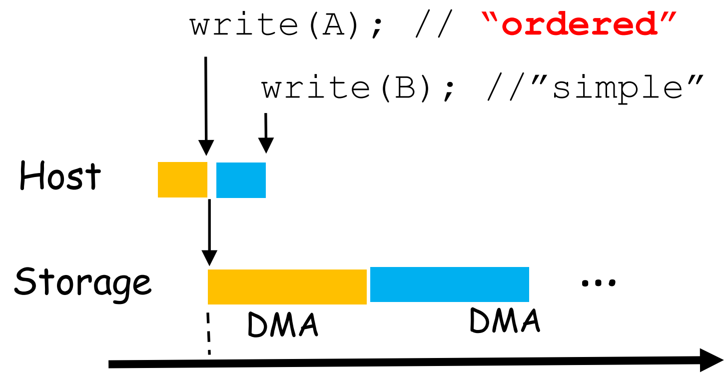
Legacy Dispatch



Caller blocks.

DMA transfer overhead

Order Preserving Dispatch



Caller does not block.

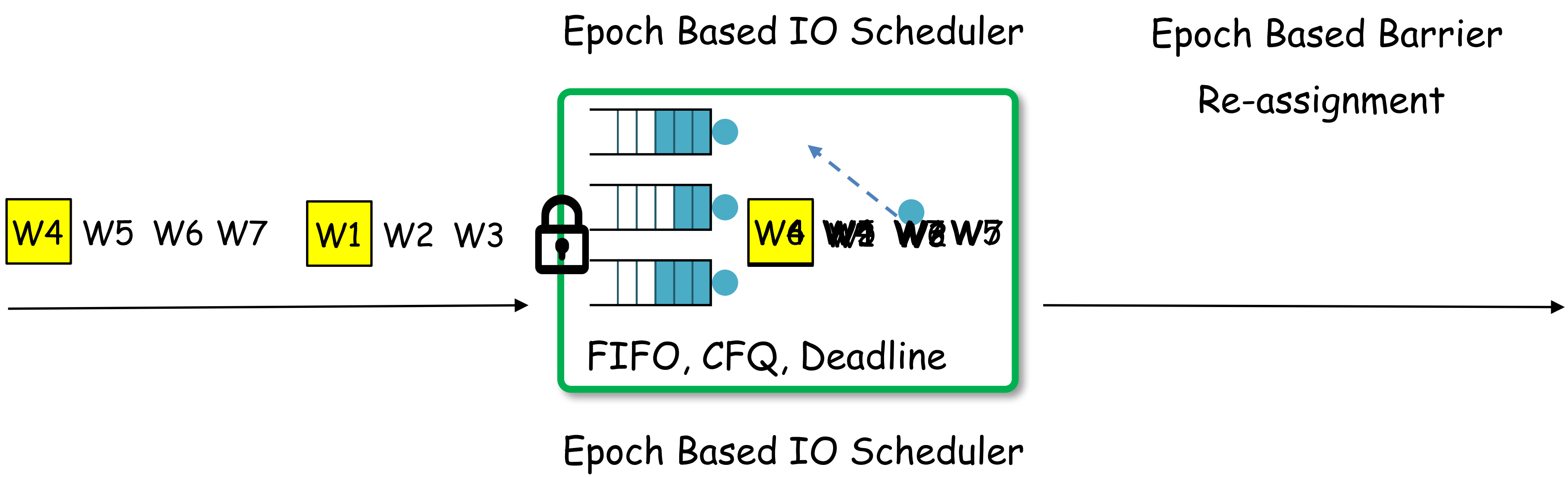
No DMA transfer overhead

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

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

Epoch Based IO scheduler (for I = D)

- Ensure that the OP requests between the barriers can be freely scheduled.
- Ensure that the OP requests does not cross barrier boundary.
- Ensure that orderless requests can be freely scheduled independent with barrier.



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

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

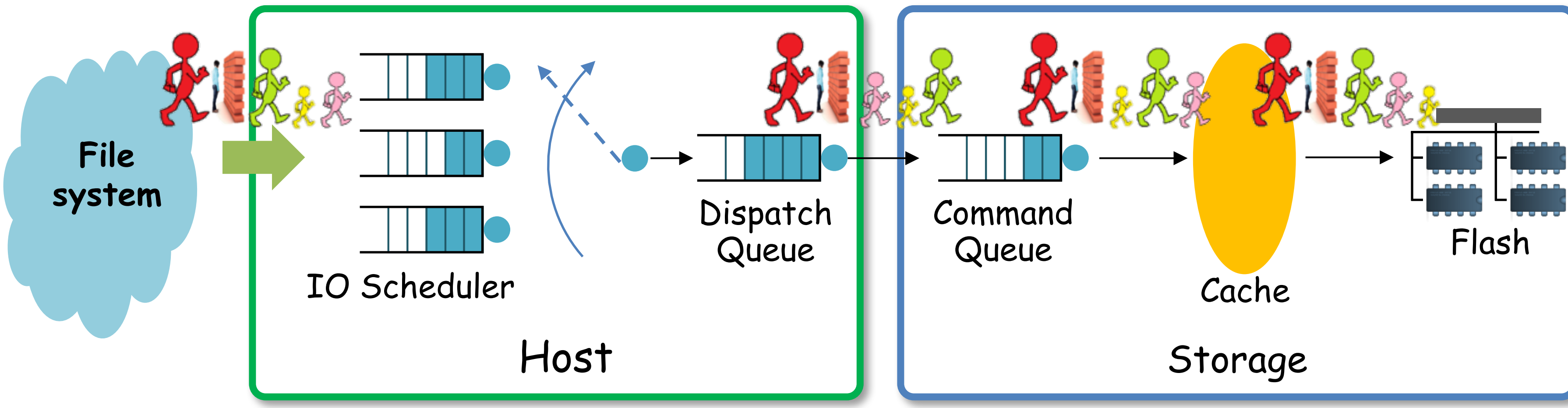
barrier write

Order-preserving dispatch

Epoch-based IO scheduler

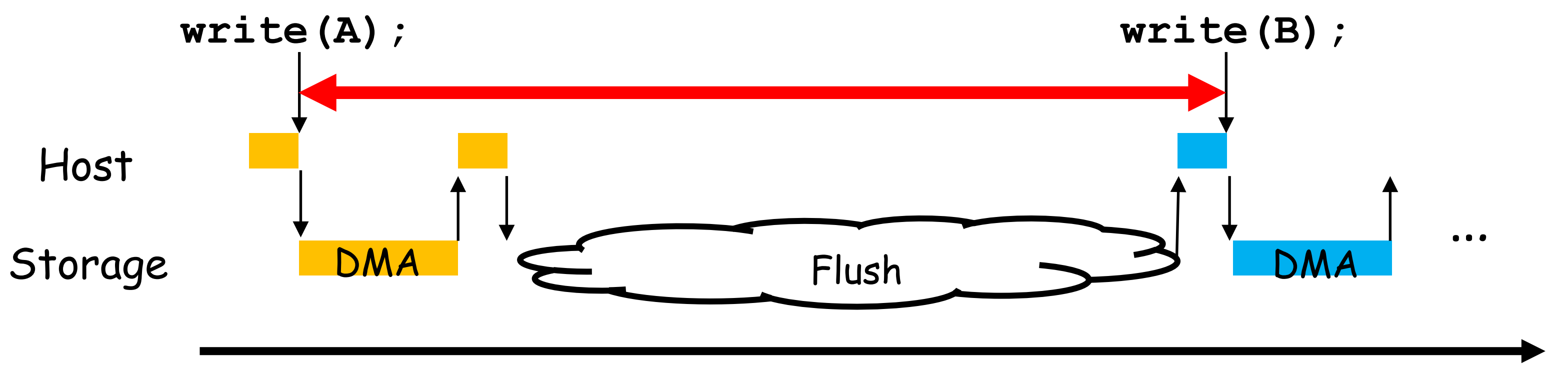
Order Preserving Block Device Layer

Control Storage Order without Transfer-and-Flush !

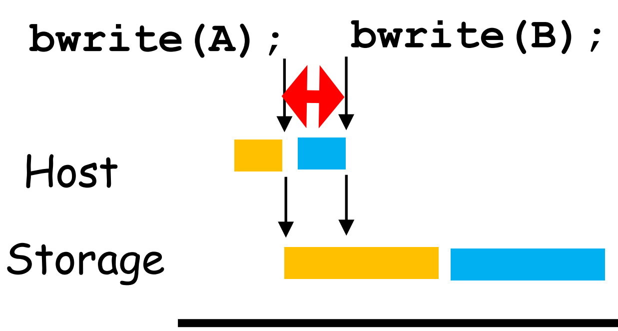


Enforcing the Storage Order

Legacy Block Layer (With Transfer-and-Flush)



Order Preserving Block Layer

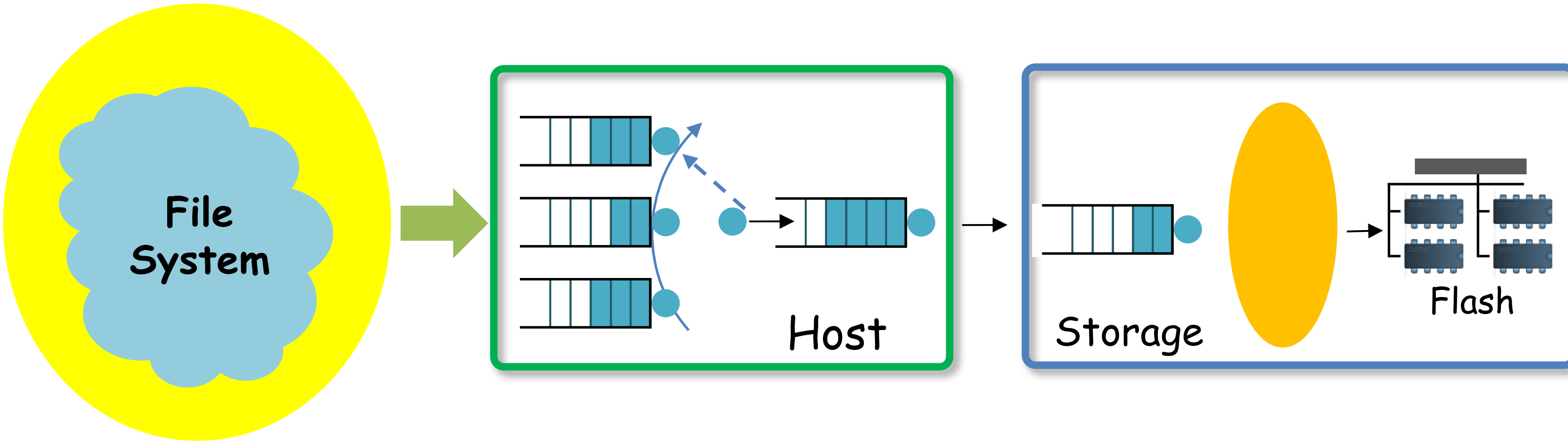


No Flush !

No DMA !

No Context Switch !

Barrier-enabled Filesystem



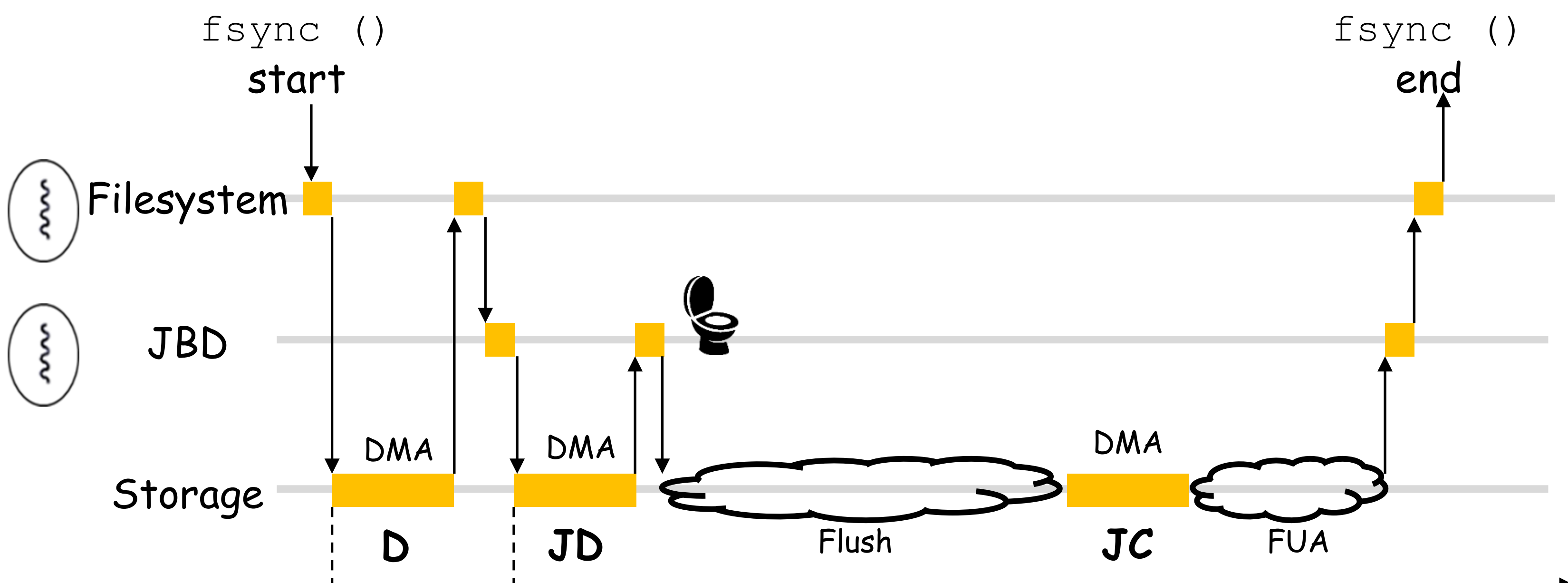
New primitives for ordering guarantee

	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~~

fsync() in EXT4

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

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



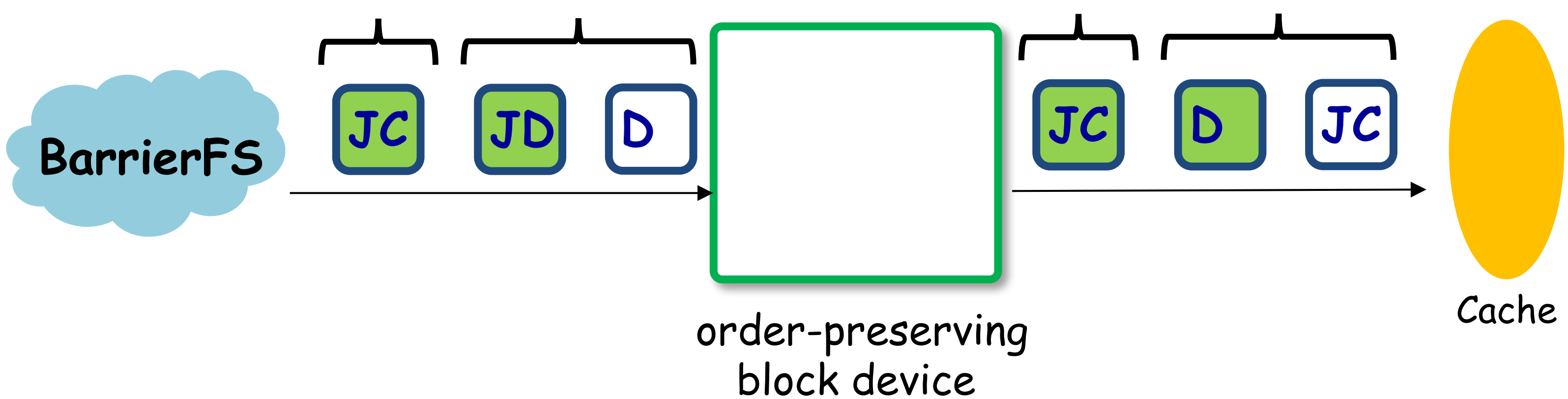
fsync() in BarrierFS

- write Dirty pages 'D' with order-preserving write
- write Journal Logs 'JD' with barrier write
- write Journal Commit Block 'JC' with barrier write
- flush

order-preserving write (REQ_ORDERED)

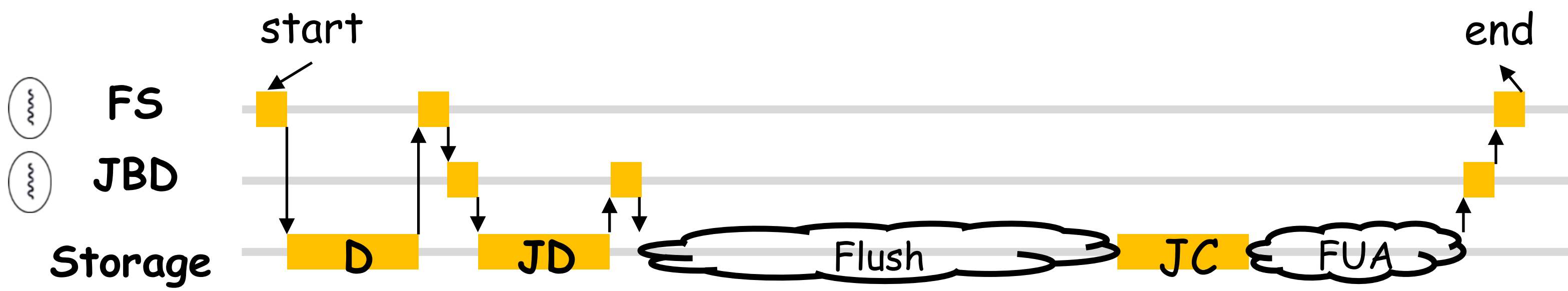
barrier write (REQ _ORDERED|REQ_BARRIER)

$\{D, JD\} \rightarrow \{JC\}$

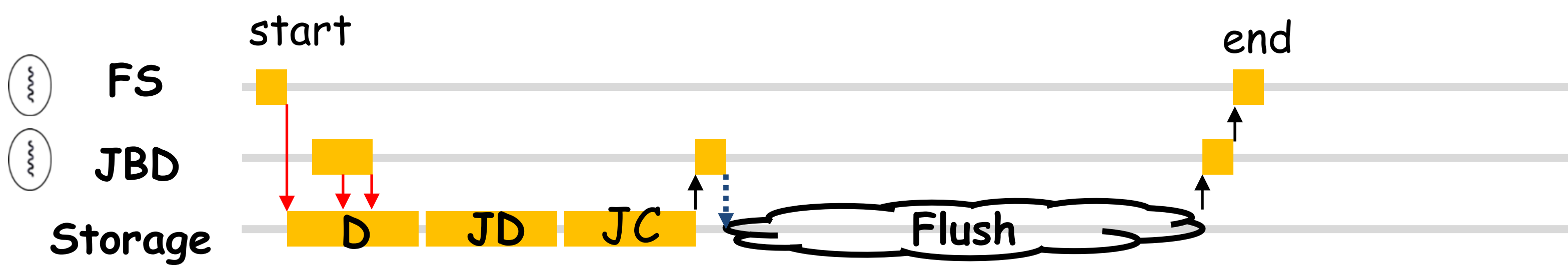


Efficient fsync() implementation

✓ fsync() in EXT4

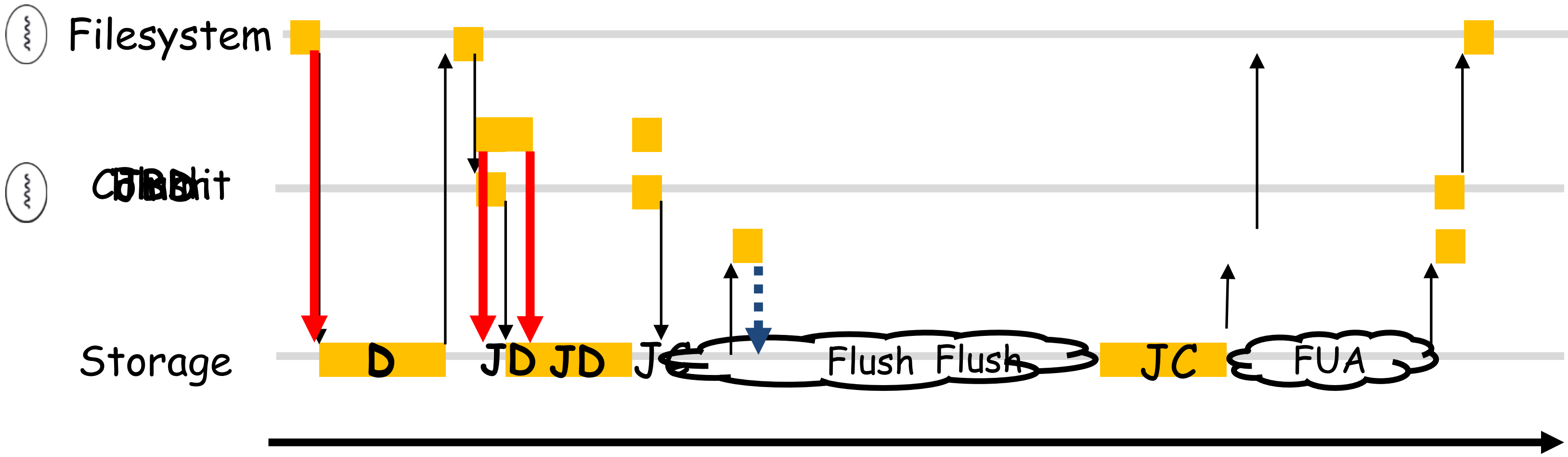


✓ fsync() in BarrierFS



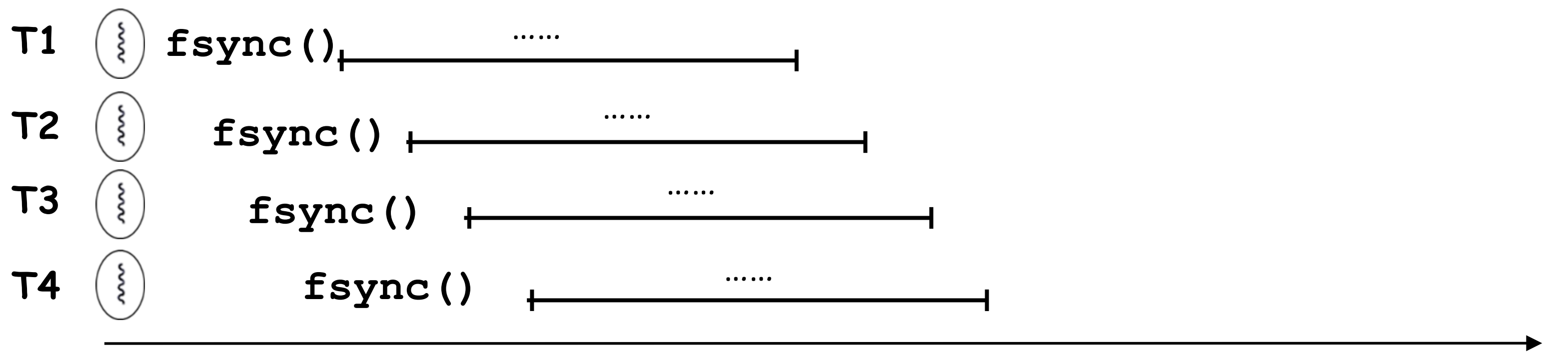
Dual Mode Journaling

- Journal Commit
  - Dispatch 'write JD' and 'write JC' → Control plane
  - Make JD and JC durable → Data Plane
- Dual Mode Journaling
  - separate the control plane activity and the data plane activity.
  - Separate thread to each
    - Commit Thread (Control Plane)
    - Flush Thread (Data Plane)

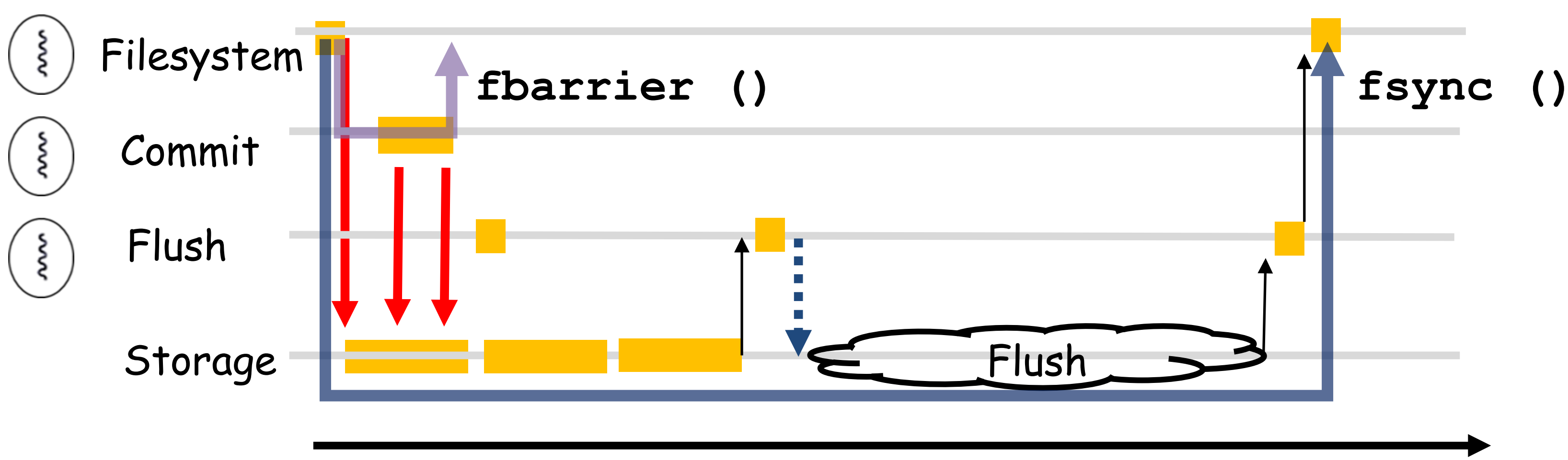


Implications of Dual Thread Journaling

- Journaling becomes concurrent activity.



- Efficient Separation of Ordering Guarantee and Durability Guarantee



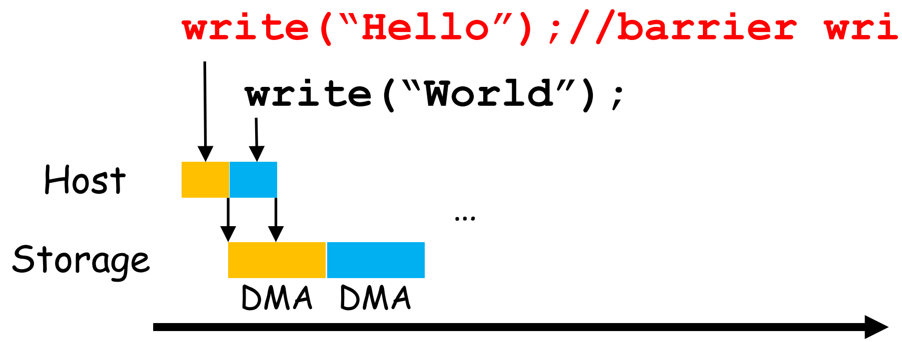
fdatabarrier()

- write Dirty pages 'D' with order-preserving write

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

BarrierFS

```
write("Hello");
write("World");
```



DMA transfer overhead NO

Flush overhead NO

Context switch NO

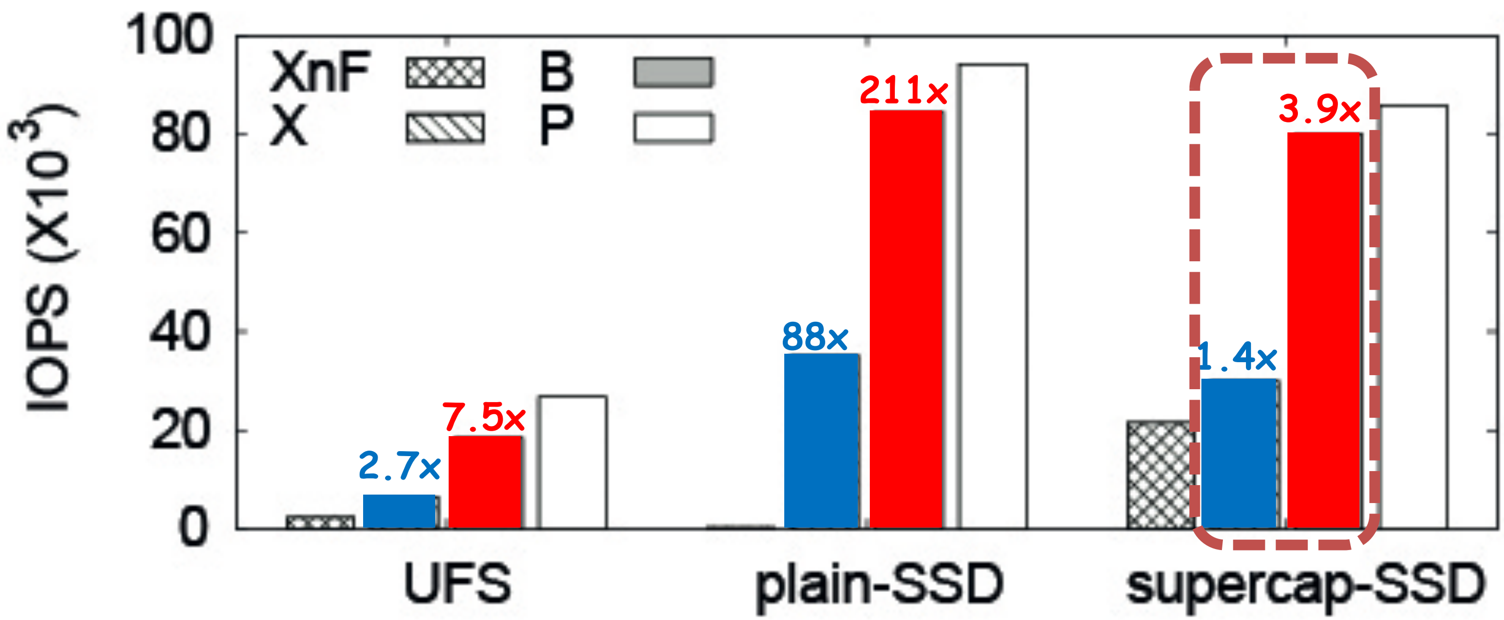
Experiments

- Platforms: PC server (Linux 3.16), Smartphone (Galaxy S6 Linux 3.10)
- Flash Storages:
  - Mobile-SSD(UFS2.0, 2ch), Plain-SSD (SM 850, 8ch), Supercap-SSD (SM843, 8ch)
- Workload
  1. Micro benchmark: Mobibench, FxMark (Microbenchmark)
  2. Macro Benchmark: Mobibench(SQLite), filebench(varmail), sysbench(MySQL)
- IO stack
  1. Durability guarantee: EXT-DR(fsync()), BFS-DR(fsync())
  2. Ordering guarantee: EXT4-OD (fdatasync(), NO-barrier), BFS-OD (fdatabarrier())

Benefit of Order-Preserving Dipspatch

Eliminating Flush

Eliminating Transfer-and-Flush

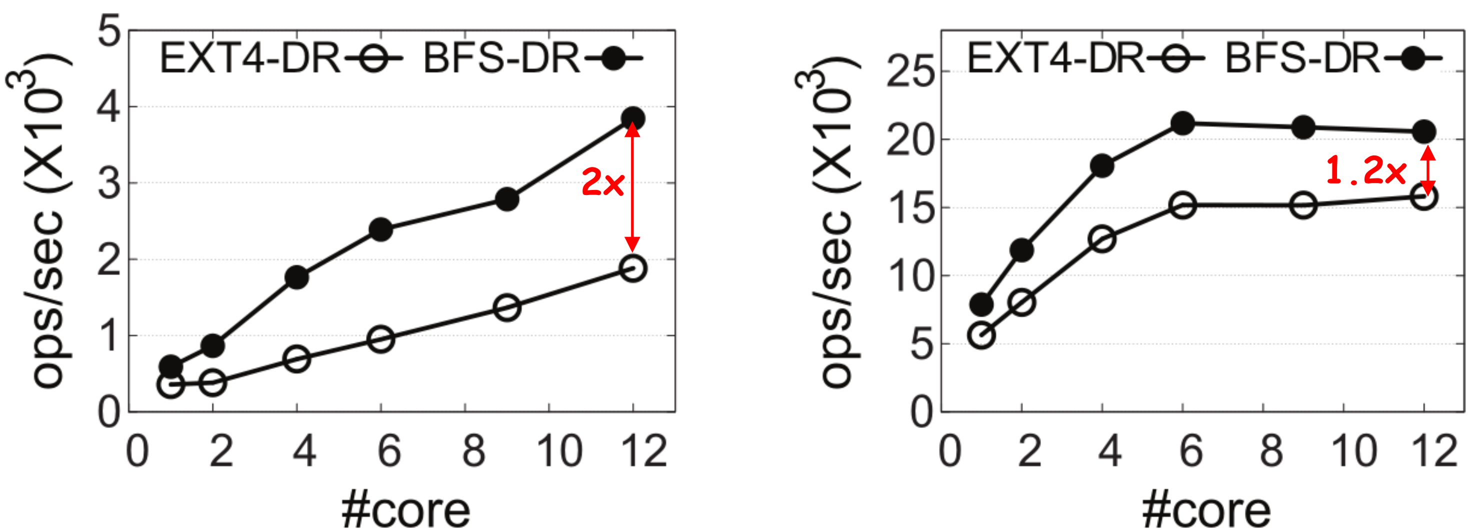


Eliminating the transfer overhead is critical.

Journaling Scalability

- 4 KB Allocating write followed by fsync() [DWSL workload in FxMark]

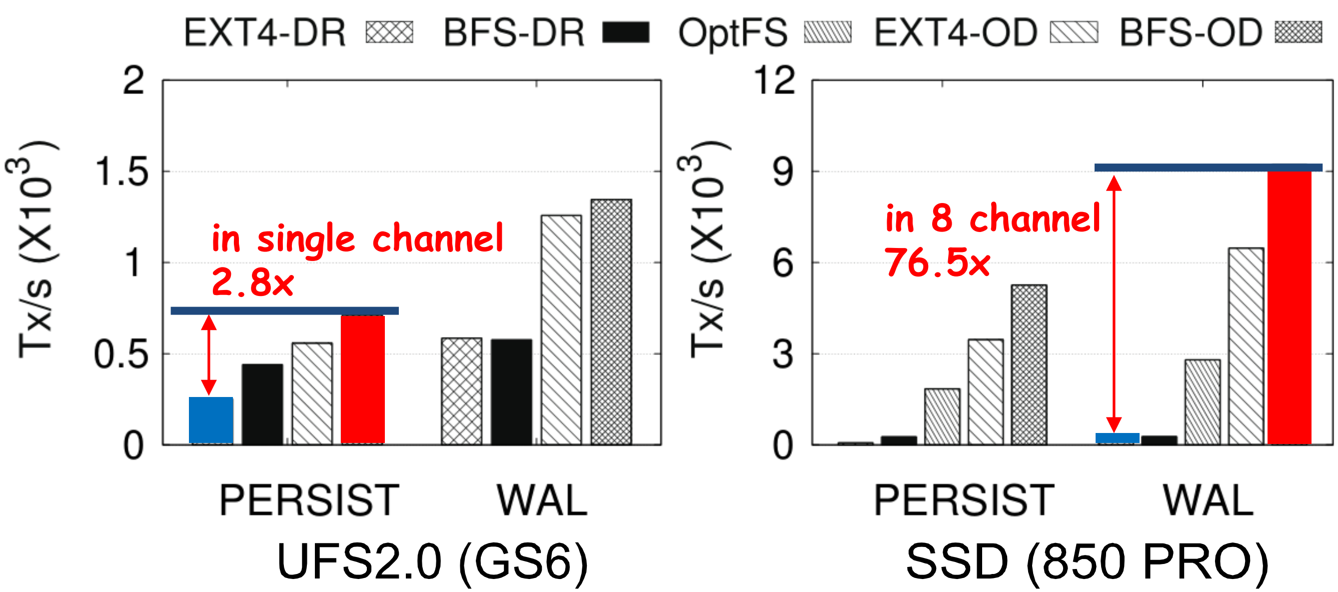
Concurrent Jounrnaling makes Journaling more scalable.



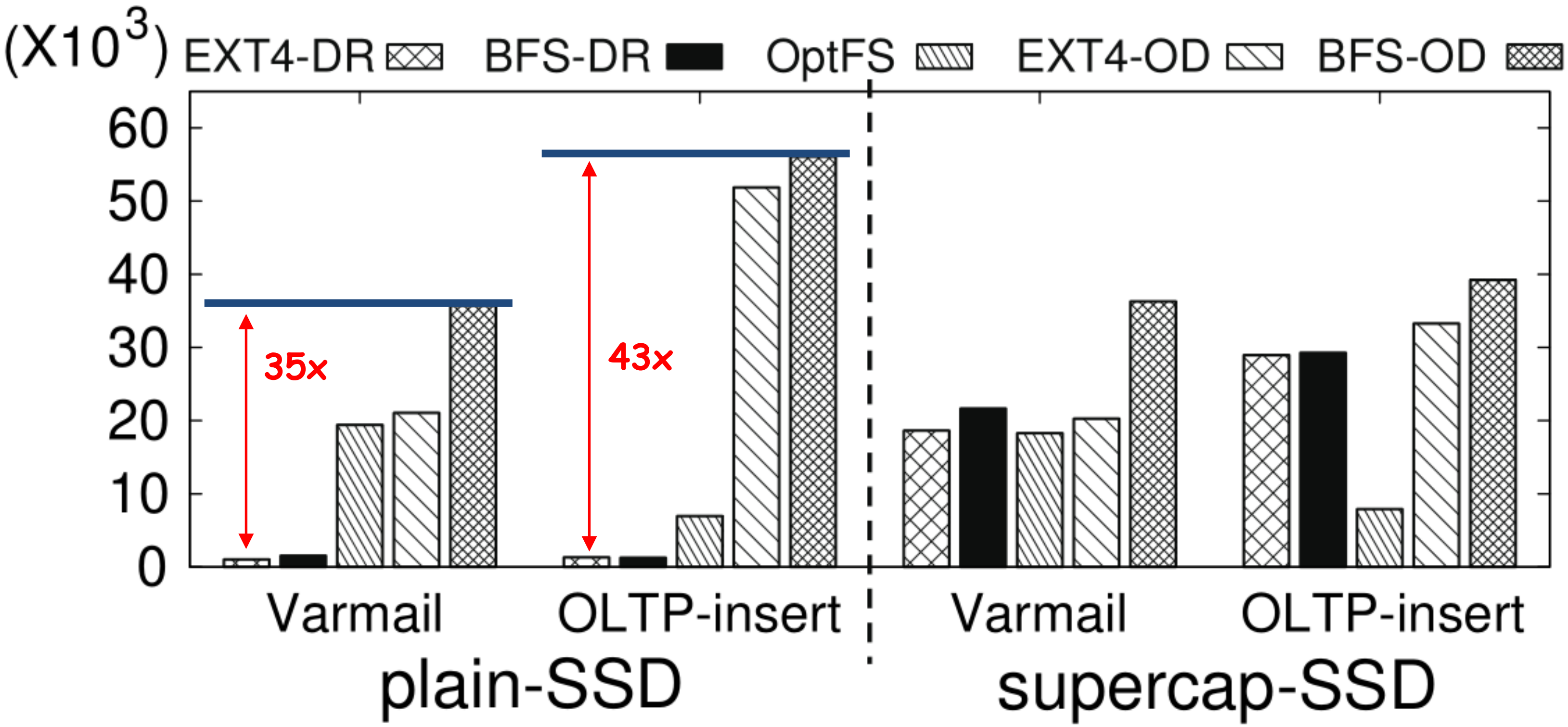
(a) plain-SSD (b) supercap-SSD

Mobile DBMS: SQLite

Barrier enabled IO stack gets more effective as the parallelism of the Flash storage increases.



Server Workload: varmail / Insert(MySQL)



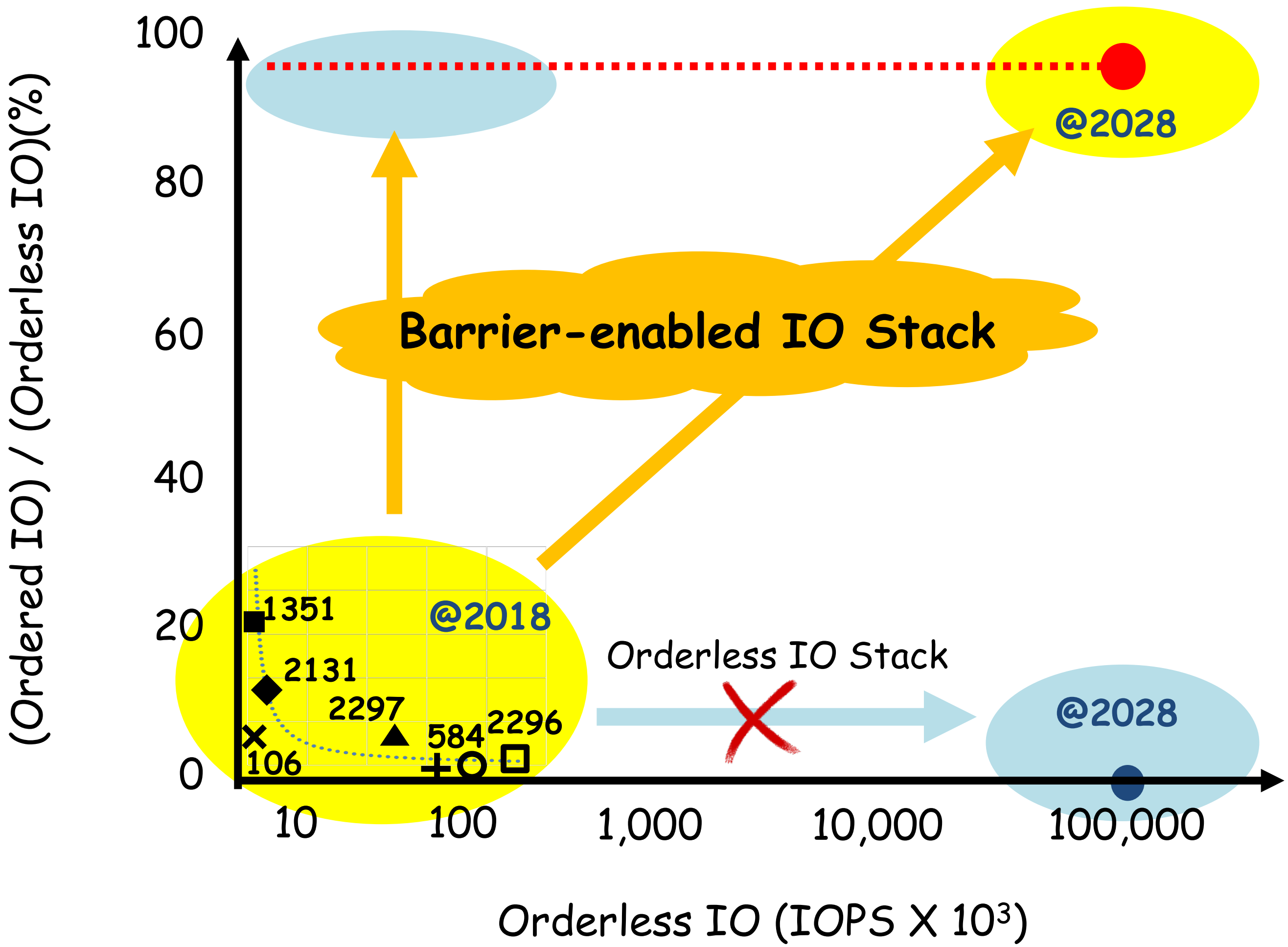
Conclusion

- Modern IO stack is fundamentally driven by the legacy of rotating media.
- In Flash Storage, the PERSIST order can be controlled while in HDD, it cannot.
- In Barrier-enabled IO stack, we eliminate the Transfer-and-Flush in controlling the storage order.
- To storage vendors,

  "Support for barrier command is a must."
- To service providers,

  "IO stack should eliminate not only the flush overhead

  but also the transfer overhead."



It is time for a change.

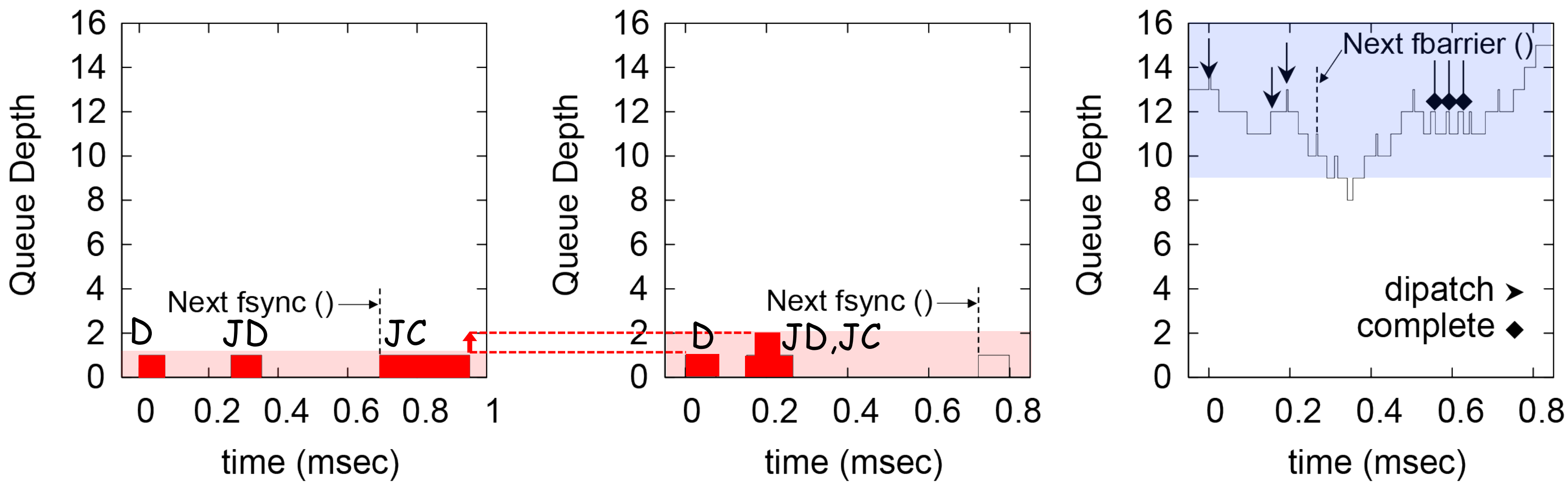
https://github.com/ESOS-Lab/barrieriostack

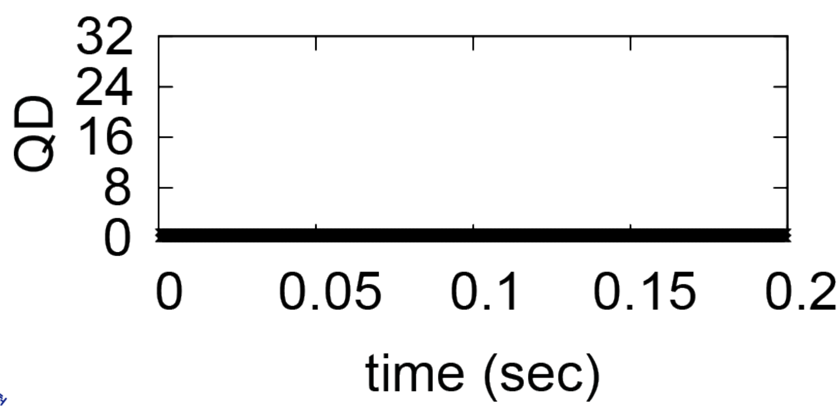
Queue Depth

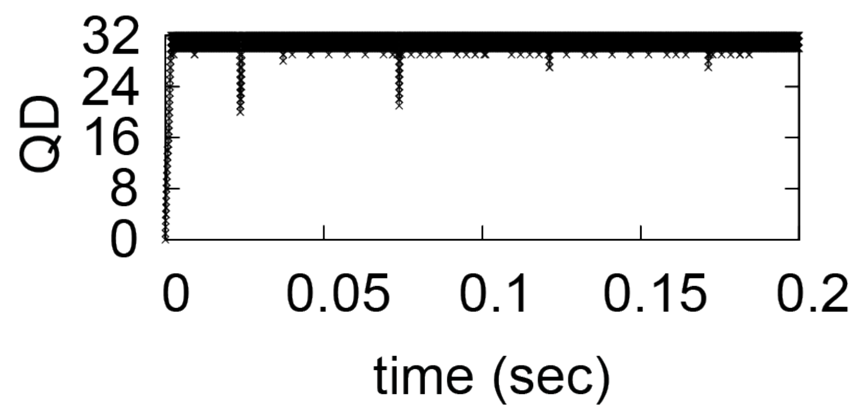
Epoch 1: {write (D), write (JD) }

Epoch 2: {write (JC)}

fsync() in EXT4 fsync() in BarrierFS fbarrier() in BarrierFS







Intel X25-M
35 K IOPS
2009

830 PRO
80 K IOPS
2012

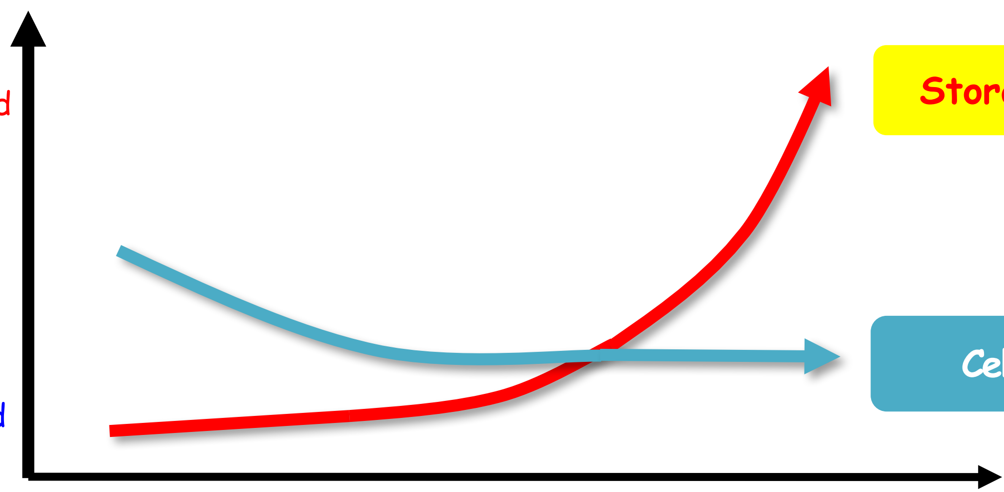
850 PRO
100 K IOPS
2014

Intel 600p
155 K IOPS
2016

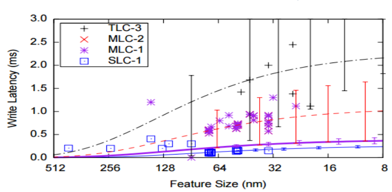
960 PRO
380 K IOPS
2016

PM1725
1 M IOPS
2015

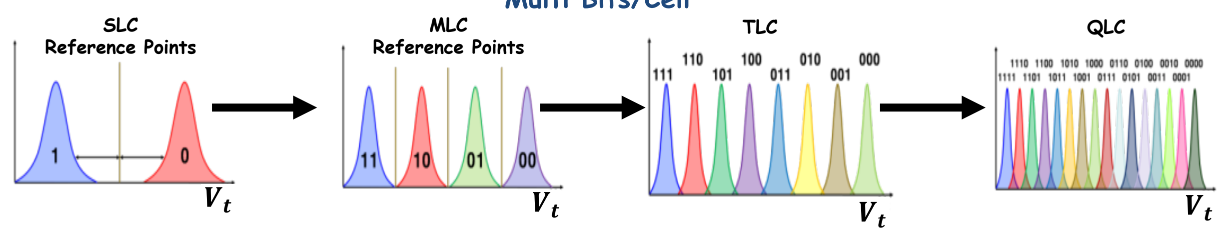
PCI Gen4
10 M IOPS
2022



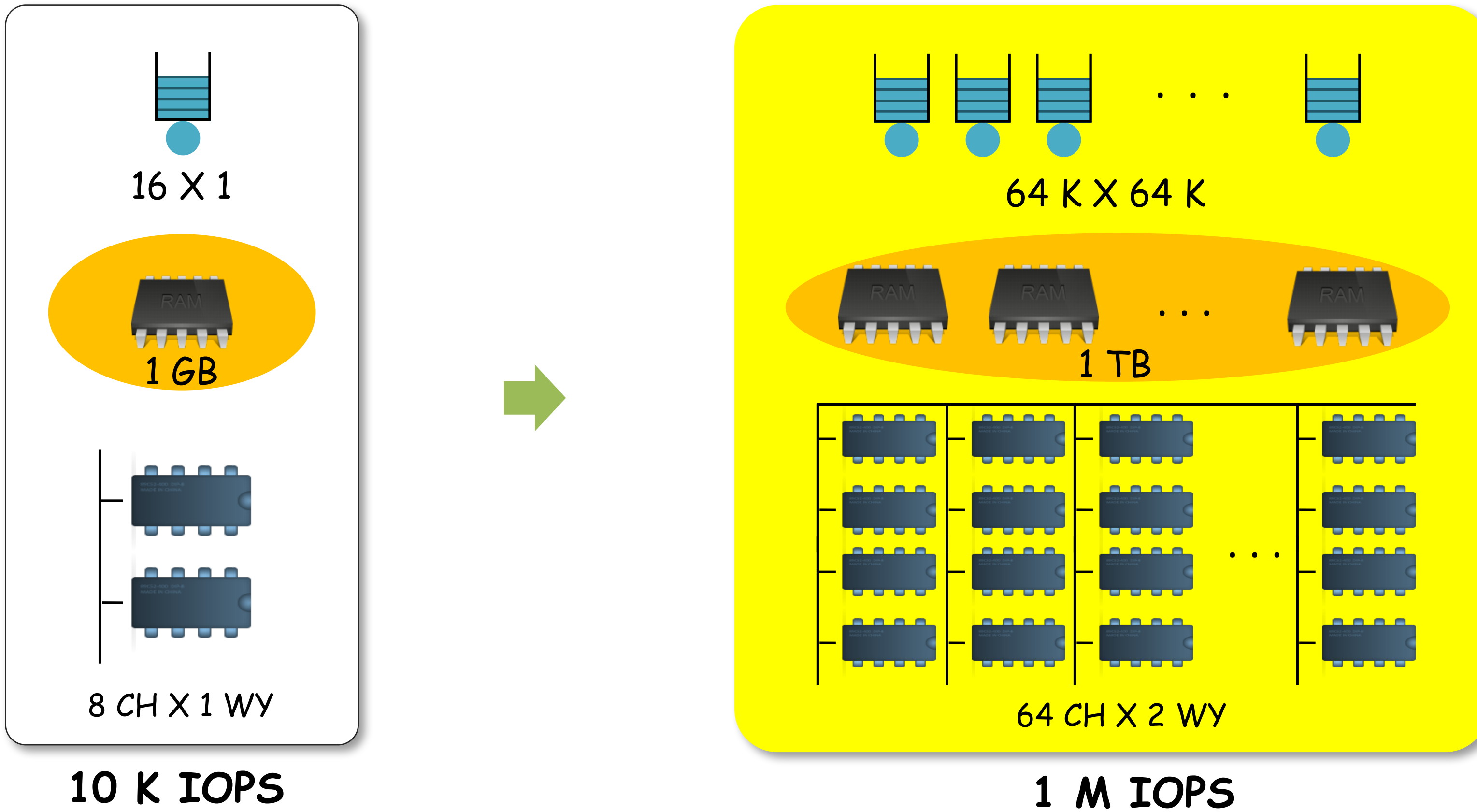
Finer Process Technology (FAST12)



Multi Bits/Cell



Storage Evolution



To Mitigate the Transfer-and-Flush overhead

- Eliminate Flush
  - Transactional checksum [IronFS,2005]
  - OptFS [2013], NoFS[2015], FeatherStitch[2007]
  - 'cache barrier'[2005], `nobarrier` option in EXT4[2010]
- Eliminate Transfer

- To reduce frequent fsync() calls
  - Log Structured Merge Tree[1996]
  - Multiple Command Queues [NVMe,2005]

Dual Mode Journaling: fbarrier()