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.

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

$I \neq D$: 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

\[(I = D) \land (D = X) \land (X = P)\]

App's -> Host -> Storage
Controlling the Storage Order

Applications need to control the storage order.

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

Logging → Commit → Checkpoint

IO scheduler, dispatch

Disk scheduler, FTL
What’s Happening Now....

Intel Optane
300 K IOPS

UFS 2.0
45K IOPS

750 NVMe
500 K IOPS

NVMe PM1725
1 M IOPS

843TN
70 K IOPS

Intel Optane
300 K IOPS

UFS 2.0
45K IOPS

750 NVMe
500 K IOPS

NVMe PM1725
1 M IOPS

843TN
70 K IOPS

Storage Vendor

I/O is Bottleneck!!!

Service Provider

Storage

facebook

Google

SQLite

CouchDB

cassandra

274 OPS/s

MySQL

1,132 OPS/s

MySQL

2,500 OPS/s

Google

facebook

SQLite

CouchDB

cassandra

274 OPS/s

MySQL

1,132 OPS/s

MySQL

2,500 OPS/s

What's Happening Now....
Overhead of storage order guarantee: write() + fdatasync()

- Galaxy S5 (eMMC 5.0)
- Samsung 950 PRO (NVMe)
- Galaxy S6 (UFS 2.0)
- OCZ RevoDrive 3 X2 (PCIe 3.0)
- Samsung 850 PRO (SATA3)
- OCZ RevoDrive 3 X2 (PCIe 3.0, RAID 0 of 4 disks)
- HDD

20% 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.

\[(I \neq P) \equiv (I = D) \land (D = X) \land (X = P)\]
Enforcing Storage Order in spite of Orderless IO Stack

Interleave the write request with Transfer-and-Flush

\[
\text{write (A) ;} \quad \rightarrow \quad \text{write (A) ;}
\]
\[
\text{write (B) ;} \quad \rightarrow \quad \text{Transfer-and-flush ;}
\]
\[
\text{write (B) ;}
\]

To enforce transfer order, block the caller!

To enforce persist order, drain the cache!
Transfer-and-Flush

Host

Storage

write(A) + fdatasync(A)

t_X

t_CS

write(B) + fdatasync(B)

t_F

\[ \Delta_{\text{transfer-and-flush}} \]

Time

App’s

Host

Storage
Overhead of Transfer-and-Flush

NVMe PM1725
120K IOPS

Ordering Guarantee
< 2%

NVMe PM1725
2K IOPS

Buffered IO (IOPS X 10^3)

(Flush IO)/(Buffered IO) (%)
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()` / `f databarrier()`

**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

\[
\begin{align*}
\text{write } &; \\
\text{barrier } &;
\end{align*}
\]

\text{single command} \rightarrow \text{barrier-write } ;

\begin{align*}
\text{write A} \\
\text{write B} \\
\text{\textbf{barrier-write C}} \\
\text{write D}
\end{align*}

\begin{align*}
\text{Persisted before} \\
\text{Persisted after}
\end{align*}
With Barrier Write command, host can control the persist order **without flush**.

\[(I \times P) \equiv (I \times D) \land (D \times X) \land (X = P)\]
Order-preserving Block Device Layer
Order Preserving Block Device Layer

✓ New request types
✓ Order Preserving Dispatch
✓ Epoch Based IO scheduling
Request Types

Orderless

{A, B, E} \rightarrow \{G, H\}

Order-Preserving

REQ_ORDERED | REQ_BARRIER

Cache

block layer
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

Diagram:
- Dispatch Queue
- Command Queue
- Cache
- Storage
SCSI Command Priority

✓ Head of the Queue

✓ Ordered (Barely being used)

✓ Simple (Default)
Order Preserving Dispatch

Legacy Dispatch

write(A); write(B);

Host

Storage

DMA

DMA...

Caller blocks.

DMA transfer overhead

Order Preserving Dispatch

write(A); // “ordered”

write(B); // “simple”

Host

Storage

DMA

DMA...

Caller does not block.

No DMA transfer overhead
With Order Preserving Dispatch, host can control the transfer order without DMA transfer.

\[(I \neq P) \equiv (I \neq D) \land (D = X) \land (X = P)\]
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.

Epoch Based IO Scheduler

Epoch Based Barrier Re-assignment

FIFO, CFQ, Deadline

Epoch Based IO Scheduler
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.
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.
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.
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.

\[(I = P) \equiv (I = D) \land (D = X) \land (X = P)\]
Order Preserving Block Device Layer

Control Storage Order without Transfer-and-Flush!
Enforcing the Storage Order

Legacy Block Layer (With Transfer-and-Flush)

write(A);
write(B);

Host

Storage

DMA

Flush

DMA

... 

Order Preserving Block Layer

bwrite(A);
bwrite(B);

Host

Storage

No Flush!
No DMA!
No Context Switch!
Barrier-enabled Filesystem
## New primitives for ordering guarantee

<table>
<thead>
<tr>
<th></th>
<th>Durability guarantee</th>
<th>Ordering guarantee</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Journaling</strong></td>
<td>✓ fsync()</td>
<td>✓ fbarrier()</td>
</tr>
<tr>
<td></td>
<td>✓ Dirty pages</td>
<td>✓ Dirty pages</td>
</tr>
<tr>
<td></td>
<td>✓ journal transaction</td>
<td>✓ Journal transaction</td>
</tr>
<tr>
<td></td>
<td>✓ Durable</td>
<td>✓ durable</td>
</tr>
<tr>
<td><strong>No journaling</strong></td>
<td>✓ fdatasync()</td>
<td>✓ fdatabarrier()</td>
</tr>
<tr>
<td></td>
<td>✓ Dirty pages</td>
<td>✓ Dirty pages</td>
</tr>
<tr>
<td></td>
<td>✓ durable</td>
<td>✓ durable</td>
</tr>
</tbody>
</table>
fsync() in EXT4

\{\text{Dirty Pages (D), Journal Logs (JD)}\} \rightarrow \{\text{Journal Commit (JC)}\}

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

![Diagram showing the process of fsync() in EXT4, including Dirty Pages (D), Journal Logs (JD), Flush, Journal Commit (JC), and DMA Transfers.]
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}

BarrierFS

order-preserving block device

Cache
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)
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

\[
\text{write}(\text{fileA}, \text{“Hello”}); \\
\text{fdatabarrier (fileA)}; \\
\text{write}(\text{fileA}, \text{“World”});
\]

\[
\text{write}(\text{“Hello”}); // \text{bwrite} \\
\text{write}(\text{“World”});
\]

\[
\text{write}(\text{“Hello”}); // \text{barrier write}
\]

write(“World”);

\[
\text{Host} \\
\text{Storage} \quad \text{DMA} \quad \text{DMA}
\]

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 Dispatch

**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 Journaling makes Journaling more scalable.

![Graphs showing performance comparison between EXT4-DR and BFS-DR for plain-SSD and supercap-SSD](image)

(a) plain-SSD  
(b) supercap-SSD
Barrier enabled IO stack gets more effective as the parallelism of the Flash storage increases.
Server Workload: varmail / Insert(MySQL)

![Graph showing performance comparison of varmail and OLTP-insert workloads between plain-SSD and supercap-SSD with different file systems.

- Varmail (plain-SSD): EXT4-DR, BFS-DR, OptFS, EXT4-OD, BFS-OD
- OLTP-insert (plain-SSD): EXT4-DR, BFS-DR, OptFS, EXT4-OD, BFS-OD

- Varmail (supercap-SSD): EXT4-DR, BFS-DR, OptFS, EXT4-OD, BFS-OD
- OLTP-insert (supercap-SSD): EXT4-DR, BFS-DR, OptFS, EXT4-OD, BFS-OD

Percentages shown:
- Varmail (plain-SSD): 35x
- OLTP-insert (plain-SSD): 43x
- Varmail (supercap-SSD): 2.5x
- OLTP-insert (supercap-SSD): 3.2x

Legend:
- EXT4-DR
- BFS-DR
- OptFS
- EXT4-OD
- BFS-OD
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