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

\[(I = D) \land (D = X) \land (X = P)\]
Controlling the Storage Order

Applications need to control the storage order.

- Database logging
- Filesystem Journaling
- Soft-updates
- COW based filesystem
What's Happening Now....

Intel Optane
300 K IOPS

UFS 2.0
45K IOPS

NVMe PM1725
1 M IOPS

843TN
70 K IOPS

NVMe
500 K IOPS

Service Provider

facebook
274 OPS/s

cassandra
2,500 OPS/s

MySQL
1,132 OPS/s

I/O is Bottleneck!!!

Storage Vendor

Storage

Queue Depth
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.

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

250MB @ 1970’s
Enforcing Storage Order in spite of Orderless IO Stack

Interleave the write request with Transfer-and-Flush

write (A) ;
write (B) ;

Transfer-and-flush;
write (B) ;

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

write(A) + fdatasync(A)

Host

write(B) + fdatasync(B)

Storage

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

Time

App’s

Host

Storage
Overhead of Transfer-and-Flush

- NVMe PM1725
  - 120K IOPS
  - Ordering Guarantee < 2%

- NVMe PM1725
  - 2K IOPS

Graph showing storage performance over time:
- Intel 750 440 K IOPS
- X25-M 35 K IOPS
- 830 PRO 80 K IOPS
- 850 PRO 100 K IOPS

Graph displaying the ratio of Flush IO to Buffered IO (%) vs. Buffered IO (IOPS x 10^3):
- Points: 1351, 2131, 2297, 584, 2296

Host to Storage diagram showing data flow and overhead.
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

\[
\begin{align*}
&\text{write} ; \\
&\text{barrier} ; \\
&\text{single command} \quad \rightarrow \quad \text{barrier-write} ;
\end{align*}
\]

- write A
- write B
- \textbf{barrier-write} C
- write D

Persisted before
Persisted after
With Barrier Write command, host can control the persist order \textit{without flush.}

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

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

Orderless

Order-Preserving

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

block layer

\begin{align*}
\text{Cache}
\end{align*}
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

SDC 2018, Santa Clara, CA
Youjip Won et al.
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.

\[(I \not= P) \equiv (I \not= D) \land (D \not= X) \land (X = P)\]
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.
With Epoch Based IO Scheduling, host can control the dispatch order with existing IO scheduler.

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

- Order-preserving dispatch
- Epoch-based IO scheduler
- barrier write
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);`
- `DMA` (between Host and Storage)
- `Flush` (indicates a flush operation)
- No `DMA` and `Context Switch`

**Order Preserving Block Layer**

- `bwrite(A);`  
- `bwrite(B);`
- `DMA` (between Host and Storage)
- No `Flush`, `DMA`, or `Context Switch`

Diagram: A flow diagram showing the processes and labels mentioned above.
Barrier-enabled Filesystem
## New primitives for ordering guarantee

<table>
<thead>
<tr>
<th></th>
<th>Durability guarantee</th>
<th>Ordering guarantee</th>
</tr>
</thead>
</table>
| **Journaling** | ![checkmark](https://www.w3.org/2000/svg/image) fsync()  
  ➢ Dirty pages  
  ➢ journal transaction  
  ➢ Durable | ![checkmark](https://www.w3.org/2000/svg/image) fbarrier()  
  ➢ Dirty pages  
  ➢ Journal transaction  
  ➢ **durable** |
| **No journaling** | ![checkmark](https://www.w3.org/2000/svg/image) fdatasync()  
  ➢ Dirty pages  
  ➢ durable | ![checkmark](https://www.w3.org/2000/svg/image) fdatabarrier()  
  ➢ Dirty pages  
  ➢ **durable** |
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
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\} → \{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)
Implications of Dual Thread Journaling

✓ Journaling becomes concurrent activity.

Efficient Separation of Ordering Guarantee and Durability Guarantee
f databarrier()

- write Dirty pages ‘D’ with order-preserving write

write(fileA, “Hello”);
f databarrier (fileA);
write(fileA, “World”);

write(“Hello”); // barrier write
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 (f databarrier())
Benefit of Order-Preserving Dispatching

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.
Mobile DBMS: SQLite

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

![Graph showing performance comparison between different file systems and flash storage channels]
Server Workload: varmail / Insert(MySQL)

(X10^3) EXT4-DR ☐ ☐ BFS-DR ☐ ☐ OptFS ☐ ☐ EXT4-OD ☐ ☐ BFS-OD ☐ ☐

plain-SSD

Varmail

OLTP-insert

supercap-SSD

Varmail

OLTP-insert

35x

43x
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)\}

\textbf{fsync() in EXT4}

\textbf{fsync() in BarrierFS}

\textbf{fbarrier() in BarrierFS}
Storage Performance

Cell Program Speed

Finer Process Technology (FAST12)

<table>
<thead>
<tr>
<th>Model</th>
<th>IOPS</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel X25-M</td>
<td>35 K</td>
<td>2009</td>
</tr>
<tr>
<td>830 PRO</td>
<td>80 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>830 PRO</td>
<td>100 K</td>
<td>2012</td>
</tr>
<tr>
<td>850 PRO</td>
<td>100 K</td>
<td>2014</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
<tr>
<td>960 PRO</td>
<td>380 K</td>
<td>2016</td>
</tr>
<tr>
<td>PM1725</td>
<td>1 M</td>
<td>2015</td>
</tr>
<tr>
<td>Intel 600p</td>
<td>155 K</td>
<td>2016</td>
</tr>
</tbody>
</table>
Storage Evolution

10 K IOPS

16 X 1

1 GB

8 CH X 1 WY

10 K IOPS

1 M IOPS

64 CH X 2 WY

64 K X 64 K

1 TB
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()