Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs

Gyusun Lee¹, Seokha Shin¹, Wonsuk Song¹, Tae Jun Ham², Jae W. Lee² and Jinkyu Jeong¹

Sungkyunkwan University (SKKU)¹
Seoul National University (SNU)²
Storage Performance Trends

- Emerging ultra-low latency SSDs deliver I/Os in a few µs

Overhead of Kernel I/O Stack

- Low-latency SSDs expose the overhead of kernel I/O stack

<table>
<thead>
<tr>
<th>Device</th>
<th>Kernel</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>SATA SSD</td>
<td>24.0%</td>
<td></td>
</tr>
<tr>
<td>NVMe SSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z-SSD</td>
<td>37.6%</td>
<td></td>
</tr>
<tr>
<td>Optane SSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SATA SSD</td>
<td>35.4%</td>
<td></td>
</tr>
<tr>
<td>NVMe SSD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z-SSD</td>
<td>35.5%</td>
<td></td>
</tr>
<tr>
<td>Optane SSD</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Read
- Write (+fsync)
Our Idea: apply asynchronous I/O concept to the I/O stack itself
Read Path Overview

Vanilla Read Path

Proposed Read Path

Latency reduction
Write Path Overview

Vanilla Write Path

```
sys_write()

Buffered write

CPU →-----------------> Device

Return to user
```
Write Path Overview

Vanilla Fsync Path

- `sys_fsync()`
- CPU → I/O stack ops. → I/O → Return to user
- Device

Proposed Fsync Path

- `sys_fsync()`
- CPU
- CPU → I/O stack ops. → I/O → Latency reduction → Return to user
- Device

Latency reduction
Agenda

• **Read path**
  – Analysis of vanilla read path
  – Proposed read path

• **Light-weight block I/O layer**

• **Write path**
  – Analysis of vanilla write path
  – Proposed write path

• **Evaluation**

• **Conclusion**
Analysis of Vanilla Read Path

- Page cache lookup: 0.30 μs
- Page allocation: 0.19 μs
- Page cache insertion: 0.33 μs
- LBA retrieval: 0.09 μs
- BIO submission: 0.72 μs
- DMA mapping: 0.29 μs
- NVMe I/O submission: 0.37 μs
- Context switch: 0.95 μs
- DMA unmapping: 0.23 μs
- Request completion: 0.81 μs
- Copy-to-user: 0.21 μs

Total latency: 12.82 μs

sys_read() → Interrupt handler

CPU → File system → Block layer → Device driver

Device → I/O submit → Interrupt

I/O 7.26 μs

Return to user
Page Allocation / DMA Mapping

Page allocation 0.19µs  DMA mapping 0.29µs

I/O submit

I/O 7.26µs

Interrupt
Asynchronous Page Allocation / DMA Mapping

• DMA-mapped page pool

![Diagram of page allocation and DMA mapping]

- Page allocation: 0.19µs
- DMA mapping: 0.29µs
- CPU
- Device
- I/O submit
- I/O: 7.26µs
- Interrupt

Core 0

Core N

64 pages

4KB DMA-mapped pages

Pagepool allocation
Asynchronous Page Allocation / DMA Mapping

- **DMA-mapped page pool**

  ![Diagram of DMA-mapped pages]

  - **Core 0**
  - **Core N**

  Pagepool allocation 0.016µs  Page allocation 0.19µs  DMA mapping 0.29µs

  - I/O submit
  - Interrupt

  I/O 7.26µs
Page Cache Insertion

Page cache tree

- Root
- Leaf node
- Page
- Page
- Page

Cache lookup?
- Miss
- Cache insertion success
- Make I/O request

Page cache lookup overhead

Page cache tree extension overhead

Prevention from duplicated I/O requests for the same file index

- Cache lookup?
  - Hit
  - Wait for page read

Page cache insertion 0.33μs

CPU

Device

I/O submit

I/O 7.26μs

Interrupt
Lazy Page Cache Insertion

Page cache tree

Root

Leaf node

Page

Page

Cache lookup?

Miss

Make I/O request

Lazy cache insertion?

Success

Duplicated I/O requests
(extremely low frequency)

Page cache lookup overhead

Page cache tree extension overhead

CPU

Device

I/O submit

I/O 7.26µs

Interrupt

Page cache insertion 0.35µs

Page free

Make I/O request

Lazy cache insertion?
DMA Unmapping

- CPU
- Device
- I/O submit
- I/O 7.26µs
- DMA unmapping 0.23µs
- Interrupt

Times:
- I/O 7.26µs
Lazy DMA Unmapping

• Implementation
  – Delays DMA unmapping to when a system is idle or waiting for another I/O requests
  – Extended version of the deferred protection scheme in Linux [ASPLOS ’16]
  – Optionally disabled for safety
Remaining Overheads in the Proposed Read Path

- BIO submission: 0.72µs
- NVMe I/O submission: 0.37µs
- Request completion: 0.81µs

I/O submit: 7.26µs

Interrupt
Agenda

• Read path
  – Analysis of vanilla read path
  – Proposed read path

• **Light-weight block I/O layer**

• Write path
  – Analysis of vanilla write path
  – Proposed write path

• Evaluation

• Conclusion
Linux Multi-queue Block I/O Layer

- **Structure conversion**
  - Merge bio with pending request via I/O merging
  - Assign new tag & request and convert from bio

- **Multi-queue structure**
  - Software staging queue (SW queue)
    ✓ Support I/O scheduling and reordering
  - Hardware dispatch queue (HW queue)
    ✓ Deliver the I/O request to the device driver

- **Multiple dynamic memory allocations**
  - Bio (block layer)
  - NVMe iod, scatter/gather list, NVMe PRP* list (device driver)

*PRP: physical region page
**Linux Multi-queue Block I/O Layer**

- **Structure conversion**
  - Inefficiency of I/O merging [Zhang, OSDI’18]
  - ✔ Useful feature for low-performance storage device

- **Multi-queue structure**
- **Multiple dynamic memory allocations**
**Linux Multi-queue Block I/O Layer**

- **Structure conversion**
  - Inefficiency of I/O merging [Zhang, _OSDI’18_]
    - Useful feature for low-performance storage device

- **Multi-queue structure**
  - Inefficiency of I/O scheduling for low-latency SSDs [Saxena, _ATC’10_] [Xu, _SYSTOR’15_]
    - Default configuration is noop scheduler
  - Bypass multi-queue structure [Zhang, _OSDI’18_]
  - Device-side I/O scheduling [Peter, _OSDI’14_] [Joshi, _HotStorage’17_]

- **Multiple dynamic memory allocations**
**Light-weight Block I/O Layer**

- **Light-weight bio (lbio) structure**
  - Contains only essential arguments for to make NVMe I/O request
  - Eliminates unnecessary structure conversions and allocations

- **Per-CPU lbio pool**
  - Supports lockless lbio object allocation
  - Supports tagging function

- **Single dynamic memory allocation**
  - NVMe PRP* list (device driver)

*PRP: physical region page
Read Path Comparison

Proposed Read Path (before applying light-weight block I/O layer)

sys_read() → BIO submission 0.72µs → NVMe I/O submission 0.37µs → Request completion 0.81µs → Return to user

CPU → I/O submit → Device → I/O 7.26µs → Interrupt

Proposed Read Path

sys_read() → LBIO submission 0.13µs → LBIO completion 0.65µs → Return to user

CPU → I/O submit → Device → I/O 7.26µs → Interrupt

Latency reduction
Read Path Comparison

Vanilla Read Path

sys_read()

CPU

Device

I/O submit

Total latency 12.82µs

I/O 7.26µs

Interrupt

Return to user

Proposed Read Path

sys_read()

CPU

Device

I/O submit

Total latency 10.10µs

I/O 7.26µs

Interrupt

Return to user

Latency reduction
Agenda

• Read path
  – Analysis of vanilla read path
  – Proposed read path

• Light-weight block I/O layer

• Write path
  – Analysis of vanilla write path
  – Proposed write path

• Evaluation

• Conclusion
Analysis of Vanilla Fsync Path (Ext4 Ordered Mode)

Data block I/O

Data block submit

Journal block submit

Flush & commit block submit

Commit block I/O

Journal block preparation
- Allocating buffer pages
- Allocating journal area block
- Checksum computation...

Commit block preparation
2.15μs

sys_fsync()

CPU

Return to user

jbd2 call 0.80μs

Journal block preparation 5.55μs

Data writeback 5.68μs

Device

Data block I/O 12.57μs

Journal block I/O 12.73μs

Commit block I/O 10.72μs
Proposed Fsync Path (Ext4 Ordered Mode)

Data Writeback 4.18μs
- Reduced latency by applying light-weight block I/O layer

sys_fsync()

jbd2 call 0.78μs
- Wake jbd2 before data block wait

Journal block preparation 5.21μs

Commit block preparation 1.90μs

jbd2 commit wait

Device

Data block I/O 11.37μs

Data block submit

Journal block submit

jbd2
Proposed Fsync Path (Ext4 Ordered Mode)

Data Writeback 4.18μs
- Reduced latency by applying light-weight block I/O layer

sys_fsync() → jbd2 call 0.78μs
- Wake jbd2 before data block wait

jbd2 commit wait

Commit block preparation 1.90μs

Journal block preparation 5.21μs

Data & journal block I/O wait
Flush & commit block dispatch 0.04μs

Data block submit

Flush & commit block submit

Device

Data block I/O 11.37μs
Journal block I/O 10.61μs
Commit block I/O 10.44μs

Return to user
Fsync Path Comparison (Ext4 Ordered Mode)

Vanilla Fsync Path

CPU → jbd2 → Device

Data block submit → Data block I/O

Journal block submit → Journal block I/O

Flush & commit block submit → Commit block I/O

Total latency 53.94μs

Latency reduction

Proposed Fsync Path

CPU → jbd2 → Device

Data block submit → Data block I/O

Journal block submit → Journal block I/O

Flush & commit block submit → Commit block I/O

Total latency 38.03μs
Agenda

• **Read path**
  – Analysis of vanilla read path
  – Proposed read path
• **Light-weight block I/O layer**
• **Write path**
  – Analysis of vanilla write path
  – Proposed write path
• **Evaluation**
• **Conclusion**
### Experimental Setup

<table>
<thead>
<tr>
<th>Server</th>
<th>Dell R730</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Ubuntu 16.04.4</td>
</tr>
<tr>
<td>Base kernel</td>
<td>Linux 5.0.5</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon E5-2640v3 2.6GHz 8-cores</td>
</tr>
<tr>
<td>Memory</td>
<td>DDR4 32GB</td>
</tr>
</tbody>
</table>
| Storage devices | Z-SSD: Samsung SZ985 800GB  
Optane SSD: Intel Optane 905P 960GB |
| Workloads       | Synthetic micro-benchmark: FIO  
Real-world workload: RocksDB DBbench |
FIO Performance (Random Read)

• **Single thread**

- Vanilla
- AIOS

Up to 23% latency reduction

- 7.6\(\mu\)s

Latency (\(\mu\)s) vs. Block size

- 4KB block size

- Vanilla
- AIOS

Up to 26% IOPS improvement

IOPS (k) vs. Threads

- Threads: 1, 2, 4, 8, 16, 32, 64, 128
FIO Performance (Random Write+Fsync, Ext4 Ordered)

- **Single thread**

  □ Vanilla ▢ AIOS

  Up to 26% latency reduction

- **4KB block size**

  □ Vanilla ▢ AIOS

  Up to 27% IOPS improvement

<table>
<thead>
<tr>
<th>Block size</th>
<th>Vanilla</th>
<th>AIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>8KB</td>
<td>15</td>
<td>12</td>
</tr>
<tr>
<td>16KB</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>32KB</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>64KB</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>128KB</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Threads</th>
<th>Vanilla</th>
<th>AIOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>40</td>
<td>32</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>
RocksDB Performance

- **DBbench readrandom**
  - 64GB dataset

  - Vanilla
  - AIOS

  ![Graph showing performance improvement up to 27%](image)

- **DBbench fillsync**
  - 16GB dataset

  - Vanilla
  - AIOS

  ![Graph showing performance improvement up to 44%](image)
Conclusion

• **Asynchronous I/O stack**
  – Applies asynchronous I/O concept to the kernel I/O stack itself
  – Overlaps computation with I/O to reduce total I/O latency

• **Light-weight block I/O layer**
  – Provides low-latency block I/O services for low-latency NVMe SSDs

• **Performance evaluation**
  – Achieves a single-digit microsecond I/O latency on Optane SSD
  – Achieves significant latency reduction and performance improvement on real-world workloads

Source code: https://github.com/skkucsl/aios
Q&A

• Thank you