FLIN:
Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives

Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose,
Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi,
Lois Orosa, Juan Gómez-Luna, Onur Mutlu

June 5, 2018
Executive Summary

- Modern solid-state drives (SSDs) use new storage protocols (e.g., NVMe) that eliminate the OS software stack
  - I/O requests are now scheduled inside the SSD
  - Enables high throughput: millions of IOPS

- OS software stack elimination removes existing fairness mechanisms
  - We experimentally characterize fairness on four real state-of-the-art SSDs
  - Highly unfair slowdowns: large difference across concurrently-running applications

- We find and analyze four sources of inter-application interference that lead to slowdowns in state-of-the-art SSDs

- FLIN: a new I/O request scheduler for modern SSDs designed to provide both fairness and high performance
  - Mitigates all four sources of inter-application interference
  - Implemented fully in the SSD controller firmware, uses < 0.06% of DRAM space
  - FLIN improves fairness by 70% and performance by 47% compared to a state-of-the-art I/O scheduler
Background: Modern SSD Design

Unfairness Across Multiple Applications in Modern SSDs

FLIN:
Flash-Level INterference-aware SSD Scheduler

Experimental Evaluation

Conclusion
**Internal Components of a Modern SSD**

- **Back End**: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)
Internal Components of a Modern SSD

- **Back End**: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)

- **Front End**: management and control units
Internal Components of a Modern SSD

- **Back End**: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)

- **Front End**: management and control units
  - **Host–Interface Logic (HIL)**: protocol used to communicate with host
- **Back End**: data storage
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)

- **Front End**: management and control units
  - Host–Interface Logic (HIL): protocol used to communicate with host
  - Flash Translation Layer (FTL): manages resources, processes I/O requests
### Internal Components of a Modern SSD

- **Back End**: data storage  
  - Memory chips (e.g., NAND flash memory, PCM, MRAM, 3D XPoint)

- **Front End**: management and control units  
  - Host–Interface Logic (HIL): protocol used to communicate with host  
  - Flash Translation Layer (FTL): manages resources, processes I/O requests  
  - Flash Channel Controllers (FCCs): sends commands to, transfers data with memory chips in back end
SSDs initially adopted conventional host–interface protocols (e.g., SATA)

- Designed for magnetic hard disk drives
- Maximum of only thousands of IOPS per device
Modern SSDs use high-performance host–interface protocols (e.g., NVMe)

- Bypass OS intervention: SSD must perform scheduling
- Take advantage of SSD throughput: enables millions of IOPS per device

Fairness mechanisms in OS software stack are also eliminated

Do modern SSDs need to handle fairness control?
Background: Modern SSD Design

Unfairness Across Multiple Applications in Modern SSDs

FLIN:
Flash-Level INterference-aware SSD Scheduler

Experimental Evaluation

Conclusion
Measuring Unfairness in Real, Modern SSDs

- We measure fairness using four real state-of-the-art SSDs
  - NVMe protocol
  - Designed for datacenters

- Flow: a series of I/O requests generated by an application

- Slowdown = \( \frac{\text{shared flow response time}}{\text{alone flow response time}} \) (lower is better)

- Unfairness = \( \frac{\max \text{ slowdown}}{\min \text{ slowdown}} \) (lower is better)

- Fairness = \( \frac{1}{\text{unfairness}} \) (higher is better)
Representative Example: \textit{tpcc} and \textit{tpce}

average slowdown of \textit{tpce}:
\textit{2x to 106x} across our four real SSDs

SSDs do not provide fairness among concurrently-running flows
What Causes This Unfairness?

- Interference among concurrently-running flows
- We perform a detailed study of interference
  - MQSim: detailed, open-source modern SSD simulator [FAST 2018]
    https://github.com/CMU-SAFARI/MQSim
  - Run flows that are designed to demonstrate each source of interference
  - Detailed experimental characterization results in the paper

- We uncover four sources of interference among flows
Source 1: Different I/O Intensities

- The I/O intensity of a flow affects the average queue wait time of flash transactions

The average response time of a low-intensity flow substantially increases due to interference from a high-intensity flow

- Similar to memory scheduling for bandwidth-sensitive threads vs. latency-sensitive threads
Some flows take advantage of chip-level parallelism in back end
Some flows take advantage of chip-level parallelism in back end.

Even distribution of transactions in chip-level queues

Leads to a low queue wait time
Other flows have access patterns that do not exploit parallelism.
Other flows have access patterns that do not exploit parallelism
Other flows have access patterns that **do not exploit parallelism**
Other flows have access patterns that do not exploit parallelism.

Flows with parallelism-friendly access patterns are susceptible to interference from flows whose access patterns do not exploit parallelism.
State-of-the-art SSD I/O schedulers prioritize reads over writes

Effect of read prioritization on fairness (vs. first-come, first-serve)

When flows have different read/write ratios, existing schedulers do not effectively provide fairness.
NAND flash memory performs **writes out of place**
- Erases can only happen on an entire flash block (hundreds of flash pages)
- Pages marked invalid during write

**Garbage collection (GC)**
- Selects a block with mostly-invalid pages
- Moves any remaining valid pages
- Erases blocks with mostly-invalid pages

**High-GC flow:** flows with a higher write intensity induce more garbage collection activities

The GC activities of a high-GC flow can unfairly block flash transactions of a low-GC flow
Summary: Source of Unfairness in SSDs

- **Four major sources of unfairness** in modern SSDs
  1. I/O intensity
  2. Request access patterns
  3. Read/write ratio
  4. Garbage collection demands

**OUR GOAL**

Design an I/O request scheduler for SSDs that
(1) provides *fairness* among flows by mitigating *all four sources of interference*, and
(2) maximizes performance and throughput
**FLIN: Flash-Level INterference-aware Scheduler**

- **FLIN** is a three-stage I/O request scheduler
  - Replaces existing transaction scheduling unit
  - Takes in flash transactions, reorders them, sends them to flash channel

- **Identical throughput to state-of-the-art schedulers**

- **Fully implemented in the SSD controller firmware**
  - No hardware modifications
  - Requires < 0.06% of the DRAM available within the SSD
Stage 1: Fairness-aware Queue Insertion relieves I/O intensity and access pattern interference

From high-intensity flows

From low-intensity flows
Stage 1: Fairness-aware Queue Insertion
relieves I/O intensity and access pattern interference
Three Stages of FLIN

- **Stage 1: Fairness-aware Queue Insertion**
  relieves I/O intensity and access pattern interference

```
Tail
9 8 7 6 5 4 3 2

Head
1 0
```
Three Stages of FLIN

- **Stage 1: Fairness-aware Queue Insertion**
  relieves I/O intensity and access pattern interference

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1</td>
<td>0 0</td>
</tr>
</tbody>
</table>

Diagram showing the flow of Flash Transactions through the stages of Fairness-aware Queue Insertion.
Three Stages of FLIN

- **Stage 1: Fairness-aware Queue Insertion**
  relieves I/O intensity and access pattern interference

- **Stage 2: Priority-aware Queue Arbitration**
  enforces priority levels that are assigned to each flow by the host
Three Stages of FLIN

- **Stage 1**: Fairness-aware Queue Insertion
  relieves I/O intensity and access pattern interference

- **Stage 2**: Priority-aware Queue Arbitration
  enforces priority levels that are assigned to each flow by the host

- **Stage 3**: Wait-balancing Transaction Selection
  relieves read/write ratio and garbage collection demand interference
Outline

Background: Modern SSD Design

Unfairness Across Multiple Applications in Modern SSDs

FLIN:
Flash-Level INterference-aware SSD Scheduler

Experimental Evaluation

Conclusion
Evaluation Methodology

MQSim: [https://github.com/CMU-SAFARI/MQSim](https://github.com/CMU-SAFARI/MQSim) [FAST 2018]

- Protocol: NVMe 1.2 over PCIe
- User capacity: 480GB
- Organization: 8 channels, 2 planes per die, 4096 blocks per plane, 256 pages per block, 8kB page size

40 workloads containing four randomly-selected storage traces

- Each storage trace is collected from real enterprise/datacenter applications: UMass, Microsoft production/enterprise
- Each application classified as low-interference or high-interference
Two Baseline Schedulers

- **Sprinkler** [Jung+ HPCA 2014]
  a state-of-the-art device-level high-performance scheduler

- **Sprinkler+Fairness** [Jung+ HPCA 2014, Jun+ NVMSA 2015]
  we add a state-of-the-art fairness mechanism to Sprinkler
  that was previously proposed for OS-level I/O scheduling
  - Does not have direct information about the internal resources and mechanisms of the SSD
  - Does not mitigate all four sources of interference
FLIN improves fairness over the baselines by an average of 70%, by mitigating all four major sources of interference.
FLIN improves performance by an average of 47%, by making use of idle resources in the SSD and improving the performance of low-interference flows.
Other Results in the Paper

- Fairness and weighted speedup for each workload
  - FLIN improves fairness and performance for *all* workloads

- Maximum slowdown
  - Sprinkler/Sprinkler+Fairness: several applications with maximum slowdown over 500x
  - FLIN: no flow with a maximum slowdown over 80x

- Effect of each stage of FLIN on fairness and performance

- Sensitivity study to FLIN and SSD parameters

- Effect of write caching
Outline

Background: Modern SSD Design

Unfairness Across Multiple Applications in Modern SSDs

FLIN:
Flash-Level INterference-aware SSD Scheduler

Experimental Evaluation

Conclusion
Modern solid-state drives (SSDs) use new storage protocols (e.g., NVMe) that **eliminate the OS software stack**
- Enables **high throughput**: millions of IOPS
- OS software stack elimination **removes existing fairness mechanisms**
- **Highly unfair slowdowns** on real state-of-the-art SSDs

**FLIN**: a new I/O request scheduler for modern SSDs designed to provide both fairness and high performance
- Mitigates all four sources of inter-application interference
  - Different I/O intensities
  - Different request access patterns
  - Different read/write ratios
  - Different garbage collection demands
- Implemented fully in the SSD controller firmware, uses < 0.06% of DRAM
- FLIN improves **fairness by 70%** and **performance by 47%** compared to a state-of-the-art I/O scheduler (Sprinkler+Fairness)
FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives

Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie S. Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, Onur Mutlu

June 5, 2018
Backup Slides
Enabling Higher SSD Performance and Capacity

- Solid-state drives (SSDs) are widely used in today’s computer systems
  - Data centers
  - Enterprise servers
  - Consumer devices

- I/O demand of both enterprise and consumer applications continues to grow

- SSDs are rapidly evolving to deliver improved performance

![Diagram of SSD architecture](image)

**Key Technologies**
- NAND Flash
- 3D XPoint
- New NVM
Defining Slowdown and Fairness for I/O Flows

- \( RT_{fi} \): response time of Flow \( f_i \)
- \( S_{fi} \): slowdown of Flow \( f_i \)

\[
S_{fi} = \frac{RT_{fi}^{shared}}{RT_{fi}^{alone}}
\]

- \( F \): fairness of slowdowns across multiple flows

\[
F = \frac{\text{MIN}_i \{S_{fi}\}}{\text{MAX}_i \{S_{fi}\}}
\]

- \( 0 < F < 1 \)
- Higher \( F \) means that system is more fair

- \( WS \): weighted speedup

\[
WS = \sum_i \frac{RT_{i}^{alone}}{RT_{i}^{shared}}
\]
Modern SSDs use **high-performance** host–interface protocols (e.g., NVMe)

- Take advantage of SSD throughput: enables *millions of IOPS* per device
- Bypass OS intervention: **SSD must perform scheduling, ensure fairness**

**Diagram:**

- Process 1
- Process 2
- Process 3

**Legend:**

- In-DRAM
- I/O Request Queue

**Inquiry:**

*Fairness should be provided by the SSD itself.*

*Do modern SSDs provide fairness?*
Flash writes can take place only to pages that are erased

- Perform **out-of-place updates** (i.e., write data to a different, free page), mark **old page as invalid**
- Update **logical-to-physical mapping** (makes use of **cached mapping table**)
- Some time later: **garbage collection** reclams invalid physical pages off the critical path of latency
Flash writes can take place only to pages that are erased
- Perform **out-of-place updates** (i.e., write data to a different, free page), mark old page as invalid
- Update **logical-to-physical mapping** (makes use of **cached mapping table**)
- Some time later: **garbage collection** reclaims invalid physical pages off the critical path of latency

**Transaction Scheduling Unit**: resolves resource contention
Motivation

- The study experimental results on our four SSDs
  - An example of two datacenter workloads running concurrently

\[\text{tpcc} \quad \text{tpce} \]

**tpce** on average experiences 2x to 106x higher slowdown compared to **tpcc**
Reason 1: Difference in the I/O Intensities

- The **I/O intensity** of a flow affects the average **queue wait time** of flash transactions.

The queue wait time highly increases with **I/O intensity**.
Reason 1: Difference in the I/O Intensities

- An experiment to analyze the effect of concurrently executing two flows with different I/O intensities on fairness
  - **Base flow**: low intensity (16 MB/s) and low average chip-level queue length
  - **Interfering flow**: varying I/O intensities from low to very high

Base flow experiences a drastic increase in the average length of the chip-level queue
An experiment to analyze the effect of concurrently executing two flows with different I/O intensities on fairness

- **Base flow**: low intensity (16 MB/s) and low average chip-level queue length
- **Interfering flow**: varying I/O intensities from low to very high

**Reason 1: Difference in the I/O Intensities**

*Base flow experiences a **drastic increase** in the average length of the chip-level queue.*

*The average response time of a low-intensity flow substantially increases due to interference from a high-intensity flow.*
**Reason 2: Difference in the Access Pattern**

- The access pattern of a flow determines how its transactions are distributed across the chip-level queues.

*Even distribution of transactions in chip-level queues*

- The running flow benefits from **parallelism** in the back end.
- Leads to a **low** transaction queue wait time.
Reason 2: Difference in the Access Pattern

- The access pattern of a flow determines how its transactions are distributed across the chip-level queues.

Uneven distribution of flash transactions

- Higher transaction wait time in the chip-level queues.
Reason 2: Difference in the Access Pattern

- An experiment to analyze the interference between concurrent flows with different access patterns
  - streaming
  - mixed

![Graph showing slowdown and fairness vs. randomness of the interfering flow]

- Base flow
- Interfering flow
- Fairness
An experiment to analyze the interference between concurrent flows with different access patterns

- Base flow: streaming access pattern (parallelism friendly)
- Interfering flow: mixed streaming and random access pattern

Reason 2: Difference in the Access Pattern

Flows with parallelism-friendly access patterns are susceptible to interference from flows with access patterns that do not exploit parallelism.
- State-of-the-art SSD I/O schedulers tend to prioritize reads over writes
  - Reads are 10-40x faster than writes
  - Reads are more likely to fall on the critical path of program execution

- The effect of read prioritization on fairness
  - Compare a first-come first-serve scheduler with a read-prioritized scheduler
Reason 3: Difference in the Read/Write Ratios

- State-of-the-art SSD I/O schedulers tend to prioritize reads over writes
  - Reads are 10-40x faster than writes
  - Reads are more likely to fall on the critical path of program execution
- The effect of read prioritization on fairness
  - Compare a first-come first-serve scheduler with a read-prioritized scheduler

Existing scheduling policies are not effective at providing fairness, when concurrent flows have different read/write ratios.
Reason 4: Difference in the GC Demands

- **Garbage collection** may block user I/O requests
  - Primarily depends on the write intensity of the workload

- An experiment with two 100%-write flows with different intensities
  - Base flow: low intensity and moderate GC demand
  - Interfering flow: different write intensities from low-GC to high-GC

  Lower fairness due to GC execution

Tries to preempt GC
Reason 4: Difference in the GC Demands

- Garbage collection may block user I/O requests
  - Primarily depends on the write intensity of the workload

- An experiment with two 100%-write flows with different intensities
  - Base flow: low intensity and moderate GC demand
  - Interfering flow: different write intensities from low-GC to high-GC

Lower fairness due to GC execution

The GC activities of a high-GC flow can unfairly block flash transactions of a low-GC flow
Stage 1: Fairness-Aware Queue Insertion

- Relieves the interference that occurs due to the **intensity** and **access pattern** of concurrently-running flows

- In concurrent execution of two flows
  - Flash transactions of one flow experience a higher increase in the chip-level queue wait time

- Stage 1 performs **reordering** of transactions within the chip-level queues to reduce the queue wait
Stage 1: Fairness-Aware Queue Insertion

From high-intensity flows

From low-intensity flows
### Stage 1: Fairness-Aware Queue Insertion

#### New Transaction Arrives

<table>
<thead>
<tr>
<th>Tail</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>Head</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

New transaction arrives
Stage 1: Fairness-Aware Queue Insertion

1. If source of the new transaction is **high-intensity**

If source of the new transaction is **low-intensity**
Stage 1: Fairness-Aware Queue Insertion

<table>
<thead>
<tr>
<th>Tail</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>Head</th>
<th>New transaction arrives</th>
</tr>
</thead>
</table>

1. If source of the new transaction is **high-intensity**

If source of the new transaction is **low-intensity**
**Stage 1: Fairness-Aware Queue Insertion**

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
<th>New transaction arrives</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

1. **If source of the new transaction is high-intensity**

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

   **If source of the new transaction is low-intensity**

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

2a. **Estimate slowdown of each transaction and reorder transactions to improve fairness in low-intensity part**

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

2b. **Estimate slowdown of each transaction and reorder transactions to improve fairness in high-intensity part**

<table>
<thead>
<tr>
<th>Tail</th>
<th>Head</th>
</tr>
</thead>
<tbody>
<tr>
<td>9 8 7 6 5 4 3 2 1 0</td>
<td>9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally 2×P read and write queues in DRAM for P priority classes.
- Stage 2
  - Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

An example:

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.
- Stage 2
  - Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

An example

```
0: [ ] [ ] [ ] [ ] [ ] [ ] [ ]
1: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
2: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally 2×P read and write queues in DRAM for P priority classes.
- Stage 2
  - Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

An example:

<table>
<thead>
<tr>
<th>Stage</th>
<th>Read Slot</th>
<th>To stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>Green</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>Blue</td>
</tr>
</tbody>
</table>

To stage 3
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally 2×P read and write queues in DRAM for P priority classes.

Stage 2
- Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3.
- It uses a weighted round-robin policy.

An example

<table>
<thead>
<tr>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1:
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.
- **Stage 2**
  - Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

**An example**

<table>
<thead>
<tr>
<th>0</th>
<th>[ ] [ ] [ ] [ ] [ ] [ ] [ ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[ ] [ ] [ ] [ ] [ ] [ ] [ ]</td>
</tr>
<tr>
<td>2</td>
<td>[ ] [ ] [ ] [ ] [ ] [ ] [ ]</td>
</tr>
</tbody>
</table>

To stage 3

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1:
  - Totally 2×P read and write queues in DRAM for P priority classes.
- Stage 2:
  - Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.
- An example

```
0: [ ] [ ] [ ] [ ] [ ]
1: [ ] [Y] [Y] [Y] [Y]
2: [ ] [ ] [ ] [ ] [ ]
```

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow
- FLIN maintains a read and a write queue for each priority level at Stage 1
  - Totally 2×P read and write queues in DRAM for P priority classes
- Stage 2
  - Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3
  - It uses a weighted round-robin policy

An example
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow
- FLIN maintains a read and a write queue for each priority level at Stage 1
  - Totally 2×P read and write queues in DRAM for P priority classes

Stage 2
- Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3
  - It uses a weighted round-robin policy

An example

<table>
<thead>
<tr>
<th></th>
<th>Read Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.
- Stage 2
  - Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

An example

<table>
<thead>
<tr>
<th>Stage</th>
<th>Read Slot</th>
<th>To stage 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally 2×P read and write queues in DRAM for P priority classes.

**Stage 2**
- Selects one ready read/write transaction from the transactions at the head of the P read/write queues and moves it to Stage 3.
- It uses a weighted round-robin policy.

**An example**

| 0 | | | | | | |
|---|---|---|---|---|---|
| 1 | | | | | | |
| 2 | | | | | | |

Read Slot
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1.
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.

Stage 2
- Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
- It uses a weighted round-robin policy.

An example

<table>
<thead>
<tr>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Read Slot → To stage 3
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different **priority levels** to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1:
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.

- **Stage 2**
  - Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
  - It uses a **weighted round-robin** policy.

- **An example**

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- **Read Slot**
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.
- FLIN maintains a read and a write queue for each priority level at Stage 1:
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.

Stage 2:
- Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
- It uses a weighted round-robin policy.

An example:

<table>
<thead>
<tr>
<th>Slot</th>
<th>Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Yellow</td>
</tr>
<tr>
<td>2</td>
<td>Blue</td>
</tr>
</tbody>
</table>

To stage 3

Read Slot
Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.

FLIN maintains a read and a write queue for each priority level at Stage 1.
- Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.

**Stage 2**
- Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
- It uses a weighted round-robin policy.

**An example**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Read Slot*
Stage 2: Priority-Aware Queue Arbitration

- Many host–interface protocols, such as NVMe, allow the host to assign different priority levels to each flow.

- FLIN maintains a read and a write queue for each priority level at Stage 1:
  - Totally $2 \times P$ read and write queues in DRAM for $P$ priority classes.

- Stage 2:
  - Selects one ready read/write transaction from the transactions at the head of the $P$ read/write queues and moves it to Stage 3.
  - It uses a weighted round-robin policy.

- An example:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To stage 3

Read Slot
Stage 3: Wait-Balancing Transaction Selection

- Minimizes interference resulted from the **read/write ratios** and **garbage collection demands** of concurrently-running flows
- Attempts to **distribute** stall times evenly across read and write transactions
- Stage 3 considers **proportional wait time** of the transactions

\[
PW = \frac{T_{\text{wait}}}{T_{\text{memory}} + T_{\text{transfer}}}
\]

*Smaller for reads*

- Reads are still **prioritized** over writes
- Reads are only prioritized when their **proportional wait time** is greater than write transaction’s proportional wait time
Stage 3: Wait-Balancing Transaction Selection

1. Estimate proportional wait times for the transactions in the read slot and write slot.
2. If the read-slot transaction has a higher proportional wait time, then dispatch it to channel.

The number of GC activities is estimated based on 1) relative write intensity, and 2) relative usage of the storage space.
**Implementation Overheads and Cost**

- FLIN can be implemented in the **firmware** of a modern SSD, and does **not require** specialized hardware.

- FLIN has to keep track of:
  - flow intensities to classify flows into and low-intensity categories,
  - slowdowns of individual flash transactions in the queues,
  - the average slowdown of each flow, and
  - the GC cost estimation data.

- **Our worst-case estimation shows that the DRAM overhead of FLIN would be very modest** (< 0.06%).

- The **maximum throughput** of FLIN is identical to the baseline:
  - All the processings are performed **off the critical path** of transaction processing.
Methodology: SSD Configuration

- **MQSim**, an open-source, accurate modern SSD simulator: [https://github.com/CMU-SAFARI/MQSim](https://github.com/CMU-SAFARI/MQSim) [FAST’18]

<table>
<thead>
<tr>
<th>SSD Organization</th>
<th>Host interface: PCIe 3.0 (NVMe 1.2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>User capacity: 480 GB</td>
<td></td>
</tr>
<tr>
<td>8 channels, 4 dies-per-channel</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flash Communication Interface</th>
<th>ONFI 3.1 (NV-DDR2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Width: 8 bit, Rate: 333 MT/s</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Flash Microarchitecture</th>
<th>8 KiB page, 448 B metadata-per-page, 256</th>
</tr>
</thead>
<tbody>
<tr>
<td>pages-per-block, 4096 blocks-per-plane, 2</td>
<td></td>
</tr>
<tr>
<td>planes-per-die</td>
<td></td>
</tr>
</tbody>
</table>

| Flash Latencies [63]              | Read latency: 75 μs, Program latency: 1300 μs, Erase latency: 3.8 ms |
Methodology: Workloads

- We categorize workloads as low-interference or high-interference
  - A workload is high-interference if it keeps all of the flash chips busy for more than 8% of the total execution time

- We form workloads using randomly-selected combinations of four low- and high-interference traces

- Experiments are done in groups of workloads with 25%, 50%, 75%, and 100% high-intensity workloads
### Methodology: Workloads

A workload is high-interference if it keeps all of the flash chips busy for more than 8% of the total execution time.

We form workloads using randomly-selected combinations of four low- and high-interference traces.

Experiments are done in groups of workloads with 25%, 50%, 75%, and 100% high-intensity workloads.

#### Table 2: Characteristics of the evaluated I/O traces.

<table>
<thead>
<tr>
<th>Trace</th>
<th>Read Ratio</th>
<th>Average Size (kB)</th>
<th>Avg. Inter Arrival (ms)</th>
<th>Interference Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>RD</td>
<td>WR</td>
<td>RD</td>
</tr>
<tr>
<td>dve [69]</td>
<td>0.68</td>
<td>21.4</td>
<td>31.5</td>
<td>4.1</td>
</tr>
<tr>
<td>eexch [68]</td>
<td>0.66</td>
<td>9.7</td>
<td>14.8</td>
<td>1.0</td>
</tr>
<tr>
<td>fin1 [7]</td>
<td>0.23</td>
<td>2.3</td>
<td>3.7</td>
<td>14.3</td>
</tr>
<tr>
<td>fin2 [7]</td>
<td>0.82</td>
<td>2.3</td>
<td>2.9</td>
<td>11.1</td>
</tr>
<tr>
<td>msnmcfs [69]</td>
<td>0.74</td>
<td>8.7</td>
<td>12.6</td>
<td>6.2</td>
</tr>
<tr>
<td>msnfs [69]</td>
<td>0.67</td>
<td>10.7</td>
<td>11.2</td>
<td>0.9</td>
</tr>
<tr>
<td>prn-0 [80]</td>
<td>0.11</td>
<td>22.8</td>
<td>9.7</td>
<td>34.2</td>
</tr>
<tr>
<td>prn-1 [80]</td>
<td>0.75</td>
<td>22.5</td>
<td>11.7</td>
<td>30</td>
</tr>
<tr>
<td>prxy-0 [80]</td>
<td>0.03</td>
<td>8.3</td>
<td>4.6</td>
<td>9.1</td>
</tr>
<tr>
<td>prxy-1 [80]</td>
<td>0.65</td>
<td>24.6</td>
<td>26.1</td>
<td>3.7</td>
</tr>
<tr>
<td>rad-be [69]</td>
<td>0.18</td>
<td>106.2</td>
<td>11.7</td>
<td>3.6</td>
</tr>
<tr>
<td>rad-ps [69]</td>
<td>0.10</td>
<td>10.3</td>
<td>8.2</td>
<td>54.2</td>
</tr>
<tr>
<td>src1-0 [80]</td>
<td>0.56</td>
<td>36.2</td>
<td>52.0</td>
<td>11.8</td>
</tr>
<tr>
<td>src1-1 [80]</td>
<td>0.95</td>
<td>35.8</td>
<td>14.7</td>
<td>3.2</td>
</tr>
<tr>
<td>src1-2 [80]</td>
<td>0.25</td>
<td>19.1</td>
<td>32.5</td>
<td>56.5</td>
</tr>
<tr>
<td>stg-0 [80]</td>
<td>0.15</td>
<td>24.9</td>
<td>9.2</td>
<td>4.6</td>
</tr>
<tr>
<td>stg-1 [80]</td>
<td>0.64</td>
<td>59.5</td>
<td>7.9</td>
<td>7.4</td>
</tr>
<tr>
<td>tpcc [68]</td>
<td>0.65</td>
<td>8.1</td>
<td>9.4</td>
<td>0.1</td>
</tr>
<tr>
<td>tpce [68]</td>
<td>0.92</td>
<td>8.0</td>
<td>12.6</td>
<td>0.1</td>
</tr>
<tr>
<td>wsrch [7]</td>
<td>0.99</td>
<td>15.1</td>
<td>8.6</td>
<td>3.0</td>
</tr>
</tbody>
</table>
For workload mixes 25%, 50%, 75%, and 100%, FLIN improves average fairness by
- 1.8x, 2.5x, 5.6x, and 54x over Sprinkler, and
- 1.3x, 1.6x, 2.4x, and 3.2x over Sprinkler+Fairness

Sprinkler+Fairness improves fairness over Sprinkler
- Due to its inclusion of fairness control

Sprinkler+Fairness does not consider all sources of interference, and therefore has a much lower fairness than FLIN
Experimental Results: Weighted Speedup

- Across the four workload categories, FLIN on average improves the weighted speedup by:
  - 38%, 74%, 132%, 156% over Sprinkler, and
  - 21%, 32%, 41%, 76% over Sprinkler+Fairness

- FLIN’s fairness control mechanism improves the performance of low-interference flows

- Weighted-speedup remains low for Sprinkler+Fairness as its throughput control mechanism leaves many resources idle
Effect of Different FLIN Stages

- The individual stages of FLIN improve **both fairness and performance** over Sprinkler, as each stage works to reduce some sources of interference.

- The fairness and performance improvements of **Stage 1** are much higher than those of **Stage 3**
  - I/O intensity is the most dominant source of interference

- **Stage 3** reduces the **maximum slowdown** by a greater amount than **Stage 1**
  - GC operations can significantly increase the stall time of transactions
- Across the four workload categories, FLIN reduces the average maximum slowdown by
  - 24x, 1400x, 3231x, and 1597x over Sprinkler, and
  - 2.3x, 5.5x, 12x, and 18x over Sprinkler+Fairness

- Across all of the workloads, no flow has a maximum slowdown greater than 80x under FLIN

- There are several flows that have maximum slowdowns over 500x with Sprinkler and Sprinkler+Fairness
Conclusion & Future Work

- FLIN is a lightweight transaction scheduler for modern multi-queue SSDs (MQ-SSDs), which provides fairness among concurrently-running flows
- FLIN uses a three-stage design to protect against all four major sources of interference that exist in real MQ-SSDs
- FLIN effectively improves both fairness and system performance compared to state-of-the-art device-level schedulers
- FLIN is implemented fully within the SSD firmware with a very modest DRAM overhead (<0.06%)

Future Work
- Coordinated OS/FLIN mechanisms