FlashShare: Punching Through Server Storage Stack from Kernel to Firmware for Ultra-Low Latency SSDs

Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Kandemir, Nam Sung Kim, Jihong Kim and Myoungsoo Jung
Executable Summary

FlashShare
Punches through the performance barriers

Latency-critical application

Reduce average **turnaround** response times by 22%

Reduce 99th-percentile **turnaround** response times by 31%

Throughput application

Unaware of ULL-SSD

Unaware of latency-critical application

Ultra low latency

Memory-like performance

Datacenter
Motivation: applications in datacenter

Datacenter executes a wide range of latency-critical workloads.
- Driven by the market of social media and web services;
- Required to satisfy a certain level of service-level agreement;
- Sensitive to the latency (i.e., turn-around response time);

A typical example: Apache

A key metric: user experience
Motivation: applications in datacenter

- Latency-critical applications exhibit varying loads during a day.
- Datacenter overprovisions its server resources to meet the SLA.
- However, it results in a low utilization and low energy efficiency.

Figure 1. Example diurnal pattern in queries per second for a Web Search cluster\(^1\).

Figure 2. CPU utilization analysis of Google server cluster\(^2\).

\(^1\)Power Management of Online Data-Intensive Services.

\(^2\)The Datacenter as a Computer.
Motivation: applications in datacenter

Popular solution: co-locating latency-critical and throughput workloads.

Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations

Micro’11

Reconciling High Server Utilization and Sub-millisecond Quality-of-Service

Eurosys’14

Heraclès: Improving Resource Efficiency at Scale

ISCA’15
Challenge: applications in datacenter

Experiment: *Apache+PageRank vs. Apache only*

Server configuration:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>i7-4790</td>
<td>Memory</td>
<td>32GB</td>
</tr>
<tr>
<td></td>
<td>3.6GHz</td>
<td>DDR3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 cores</td>
<td>Chipset</td>
<td>H97</td>
</tr>
</tbody>
</table>

Applications:
Apache – Online latency-critical application;
PageRank – Offline throughput application;

Performance metrics:
SSD device latency;
Response time of latency-critical application;
**Challenge**: applications in datacenter

**Experiment**: *Apache+PageRank vs. Apache only*

- The throughput-oriented application drastically increases the I/O access latency of the latency-critical application.
- This latency increase deteriorates the turnaround response time of the latency-critical application.

**Fig 1**: Apache SSD latency increases due to PageRank.

**Fig 2**: Apache response time increases due to PageRank.
Challenge: ULL-SSD

There are emerging Ultra Low-Latency SSD (ULL-SSD) technologies, which can be used for faster I/O services in the datacenter.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Optane</th>
<th>nvNitro</th>
<th>ZNAND</th>
<th>XL-Flash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase change</td>
<td>Phase change</td>
<td>MRAM</td>
<td>New NAND Flash</td>
<td></td>
</tr>
<tr>
<td>RAM</td>
<td>RAM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vendor</td>
<td>Intel</td>
<td>Everspin</td>
<td>Samsung</td>
<td>Toshiba</td>
</tr>
<tr>
<td>Read</td>
<td>10us</td>
<td>6us</td>
<td>3us</td>
<td>N/A</td>
</tr>
<tr>
<td>Write</td>
<td>10us</td>
<td>6us</td>
<td>100us</td>
<td>N/A</td>
</tr>
</tbody>
</table>
In this work, we use engineering sample of Z-SSD.

<table>
<thead>
<tr>
<th>Technology</th>
<th>SLC based 3D NAND 48 stacked word-line layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Capacity</td>
<td>64Gb/die</td>
</tr>
<tr>
<td>Page Size</td>
<td>2KB/Page</td>
</tr>
</tbody>
</table>

Challenge: Datacenter server with ULL-SSD

Unfortunately, the short latency characteristics of ULL-SSD cannot be exposed to users (in particular, for the latency-critical applications).

Server configuration:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>i7-4790 3.6GHz 8 cores</td>
<td>Memory</td>
<td>32GB DDR3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chipset</td>
<td>H97</td>
</tr>
</tbody>
</table>

Applications:

*Apache* – online latency-critical application;

*PageRank* – offline throughput application;
**Challenge:** Datacenter server with ULL-SSD

ULL-SSD fails to bring short latency, because of the storage stack.

The current design of blkmq layer, NVMe driver, and SSD firmware can hurt the performance of latency-critical applications.

The storage stack is unaware of the characteristics of both latency-critical workload and ULL-SSD.
**Blkmq layer: challenge**

Software queue: holds latency-critical I/O requests for a long time;
Software queue: holds latency-critical I/O requests for a long time;
Hardware queue: dispatches an I/O request without a knowledge of the latency-criticality;
**Blkmq layer: optimization**

Our solution bypass blkmq for a faster response.

Throughput I/Os: merge in blkmq for a higher storage bandwidth.

---

Diagram:
- **App** requests
- **Caching layer**
- **Filesystem**
- **blkmq**
- **NVMe Driver**
- **ULL-SSD**

**I/O submission**

**Incoming requests**:
- LatReq
- ThrReq

**Bypass**

**LatReq**

**Throughput I/Os**:
- Merge in blkmq for a higher storage bandwidth.

**Latency-critical I/Os**:
- Bypass blkmq for a faster response.

**Little penalty**

**Addressed in NVMe**
NVMe SQ: challenge (bypass is not simple enough)

NVMe protocol-level queue: a latency-critical I/O request can be blocked by prior I/O requests;

\[ \text{Time Cost} = T_{\text{fetch-self}} + 2xT_{\text{fetch}} \]

>200% overhead
**Target:** Designing towards a responsiveness-aware NVMe submission.

**Key Insight:**
- Conventional NVMe controller(s) allow to customize the standard arbitration strategy for different NVMe protocol-level queue accesses.
- Thus, we can make the NVMe controller to decide which NVMe command to fetch by sharing a hint for the I/O urgency.
NVMe SQ: optimization

Our Solution:

1. Double SQs (one for latency-critical I/Os, another for throughput I/Os)
2. Double the SQ doorbell register
3. New arbitration strategy: gives the highest priority to the Lat-SQ

Incoming requests:
- ThrReq
- LatReq

Diagram:
- Caching layer
- Filesystem
- blkmq
- NVMe Driver
- ULL-SSD
- NVMe SQ doorbell
- Lat-SQ
- Thr-SQ
- CQ
- Core 0
- Core 1
- Core 2
- Lat-SQ doorbell
- Thr-SQ doorbell
- Immediate fetch
- Postpone
- Core 0
- Core 1
- Core 2
- I/O submission
- Application
- Core 0
- Core 1
- Core 2
- NVMe CQ
- NVMe SQ
- NVMe Driver
- ULL-SSD
- Caching layer
- Filesystem
- blkmq
SSD firmware: challenge

Embedded cache can be polluted by the throughput part of I/O requests;
(DRAM service)

Incoming requests

Cost: $T_{CL} + T_{CACHE} + T_{NAND} + T_{CACHE}$

<table>
<thead>
<tr>
<th>way</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x4</td>
</tr>
<tr>
<td>1</td>
<td>0x8</td>
</tr>
<tr>
<td>2</td>
<td>0xb</td>
</tr>
</tbody>
</table>

Evict

0x0
0x1
0x5
SSD firmware: optimization

Our design: splits the internal cache space to protect latency-critical I/O requests;

Incoming requests:
- Req@0x01
- Req@0x05
- Req@0x04
- Req@0x08

I/O submission:
- App
- App
- App

Embedded Cache:

<table>
<thead>
<tr>
<th>way</th>
<th>addr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x8</td>
</tr>
<tr>
<td>1</td>
<td>0x1</td>
</tr>
<tr>
<td>2</td>
<td>0x5</td>
</tr>
</tbody>
</table>

Evict: 0x4
Protection region: 0x0
NVMe CQ: challenge

NVMe completion: MSI overhead for each I/O request:

Cost: $T_{CS}$

Cost: $T_{ISR}$

Cost: $2 \times T_{CS}$

I/O completion

MSI Interrupt

Interrupt Controller

CPU context switch

Interrupt Service Routine context switch

NVMe SQ

NVMe CQ

Core 0

Core 1

Core 2

SQ doorbell register

Message controller

CQ doorbell register

ULL-SSD

NVMe Driver

Caching layer

Filesystem

blkmq

App

App

App

Message

Interrupt Controller

CPU context switch

Interrupt Service Routine context switch

NVMe SQ

NVMe CQ

Core 0

Core 1

Core 2

SQ doorbell register

Message controller

CQ doorbell register

ULL-SSD

NVMe Driver

Caching layer

Filesystem

blkmq

App

App

App
Key insight: state-of-the-art Linux supports a poll mechanism;
NVMe CQ: optimization

Poll mechanism can bring benefits to fast storage device.

ULL SSD

Average Latency (µsec)

4KB 8KB 16KB 32KB

Read

Interupt

Polling

Write

Interupt

Polling

Decreases by
Read: 7.5% & Write: 13.2%
However, the poll-based I/O services consume most host resources.
**NVMe CQ: optimization**

**Our solution**: selective interrupt service routine (*Select_ISR*).

---

**Diagram Description**:
- **NVMe Driver**: Receives I/O completion requests from ULL-SSD.
- **Caching layer**: Processes incoming requests before passing them to Filesystem.
- **Filesystem**: Handles high-level file system operations.
- **blkmq**: Provides a low-level queueing mechanism.
- **Interrupt Controller**: Manages interrupt requests.
- **LatReq**: Latency requests handled by the controller.
- **CQ doorbell register**:NVMe controller communicates with the CPU through this register.
- **SQ doorbell register**:NVMe SQ register.
- **MSI Interrupt**: Triggered by NVMe Driver.
- **Interrupt Routine**: Processes hardware interrupts.
- **CPU**: Performs context switch.
- **Core 0, Core 1, Core 2**: CPU cores handling the Interrupt Routine.
- **Sleep**: Indicating CPU state during sleep.
**Design:** Responsiveness Awareness

**Key Insight:** users have a better knowledge of I/O responsiveness (i.e., latency critical/throughput).

**Our Approach:**
- Open a set of APIs to users, which pass the workload attribute to Linux PCB.

```c
int sched_setworkload_attr(){
    ...
    return syscall(
        sys_sched_setworkload_attr, pid, attr);
} 

int sched_getworkload_attr(){
    ...
    return syscall(
        sys_sched_getworkload_attr, pid, attr);
}
```

Call a new utility: `chworkload_attr`

**Usage:** `chworkload_attr -t type [-p process_id] [user_program]`

**Description:** set application type
**Key Insight:** users have a better knowledge of I/O responsiveness (i.e., latency critical/throughput).

**Our Approach:**
- Open a set of APIs to users, which pass the workload attribute to Linux PCB.
- Deliver the workload attribute to each layer of storage stack.
More optimizations

Advanced caching layer designs:
• Dynamic cache split scheme: to maximize cache hits in various request patterns;
• Read prefetching: better utilize SSD internal parallelism;
• Adjustable read prefetching with ghost cache: adaptive to different request patterns;

Hardware accelerator designs:
• Conduct simple but timing-consuming tasks such as I/O poll and I/O merge;
• Simplify the design of blkmq and NVMe driver.
Experiment Setup

Test Environment

<table>
<thead>
<tr>
<th>gem5 parameters</th>
<th>value</th>
<th>SimpleSSD parameters</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>core</td>
<td>64-bit ARM, 8, 2GHz</td>
<td>read/write/erase</td>
<td>3us/100us/1ms</td>
</tr>
<tr>
<td>L1D$/L1I$</td>
<td>64KB, 64KB</td>
<td>channel/package</td>
<td>16/1</td>
</tr>
<tr>
<td>mem controller</td>
<td>1</td>
<td>die/plane</td>
<td>8/8</td>
</tr>
<tr>
<td>memory</td>
<td>DDR3, 2GB</td>
<td>page size</td>
<td>2KB</td>
</tr>
<tr>
<td>Kernel</td>
<td>4.9.30</td>
<td>DMA/PCIe</td>
<td>800MHz, 3.0, x4</td>
</tr>
<tr>
<td>Image</td>
<td>Ubuntu 14.04</td>
<td>DRAM cache</td>
<td>1.5GB</td>
</tr>
</tbody>
</table>

System configurations:
- Vanilla – a vanilla Linux-based computer system running on ZSSD;
- CacheOpt – compared to Vanilla, it optimizes the cache layer of SSD firmware;
- KernelOpt – it optimizes blkmq layer and NVMe I/O submission;
- SelectISR – compared to KernelOpt, it adds the optimization of selective ISR;

http://simplessd.org
**Evaluation: latency breakdown**

- **KernelOpt** reduces the time cost of blkmq layer by 46% thanks to no queuing time;
- As latency-critical I/Os are fetched by NVMe controller immediately, **KernelOpt** drastically reduces the waiting time;
- **CacheOpt** better utilizes the embedded cache layer and reduces the SSD access delays by 38%;
- By selectively using polling mechanism, **SelectISR** can reduce the I/O completion time by 5us.
Evaluation: online I/O access

- **CacheOpt** reduces the average I/O service latency, but it cannot eliminate the long tails;
- **KernelOpt** can remove the long tails, because it can avoid long queuing time and prevents throughput I/Os from blocking latency-critical I/Os;
- **SelectISR** reduces the average latency further, thanks to selectively using poll mechanism.
Conclusion

**Observation**
The ultra-low latency of new memory-based SSDs is not exposed to latency-critical application and have no benefit from user-experience angle;

**Challenge**
Piecemeal reformations of the current storage stack won’t work due to multiple barriers; the storage stack is unaware of all behaviors of ULL-SSD and latency-critical applications;

**Our solution**
FlashShare: We expose different levels of I/O responsiveness to the key components in the current storage stack and optimize the corresponding system layers to make ULL visible to users (latency-critical applications).

**Major results**
- Reducing average turnaround response times by 22%;
- Reducing 99th-percentile turnaround response times by 31%.
Thank you