New Interface Design for New NVRAM Storage

Heon Young YEOM

Seoul National University

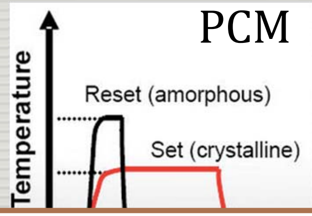
New NVRAM Storage Systems

HDD

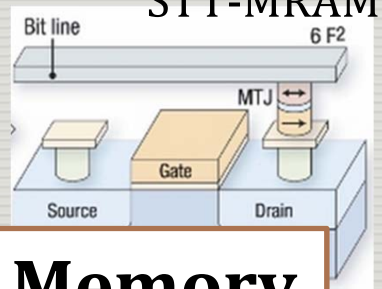
Response Time =
Seek (10ms)
+ Rotational Delay (0.8ms)
+ Transfer Time

Flash Memory

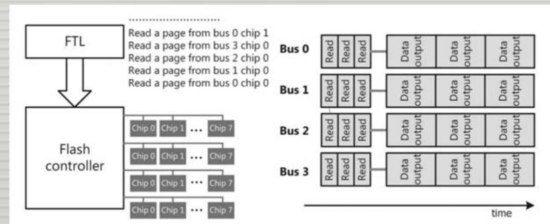
PCM



STT-MRAM



Storage Class Memory



Ozone(O3): An out-of-order Flash memory Controller Architecture, IEEE Trans. On Computes, May 2011.

Parallel Architecture

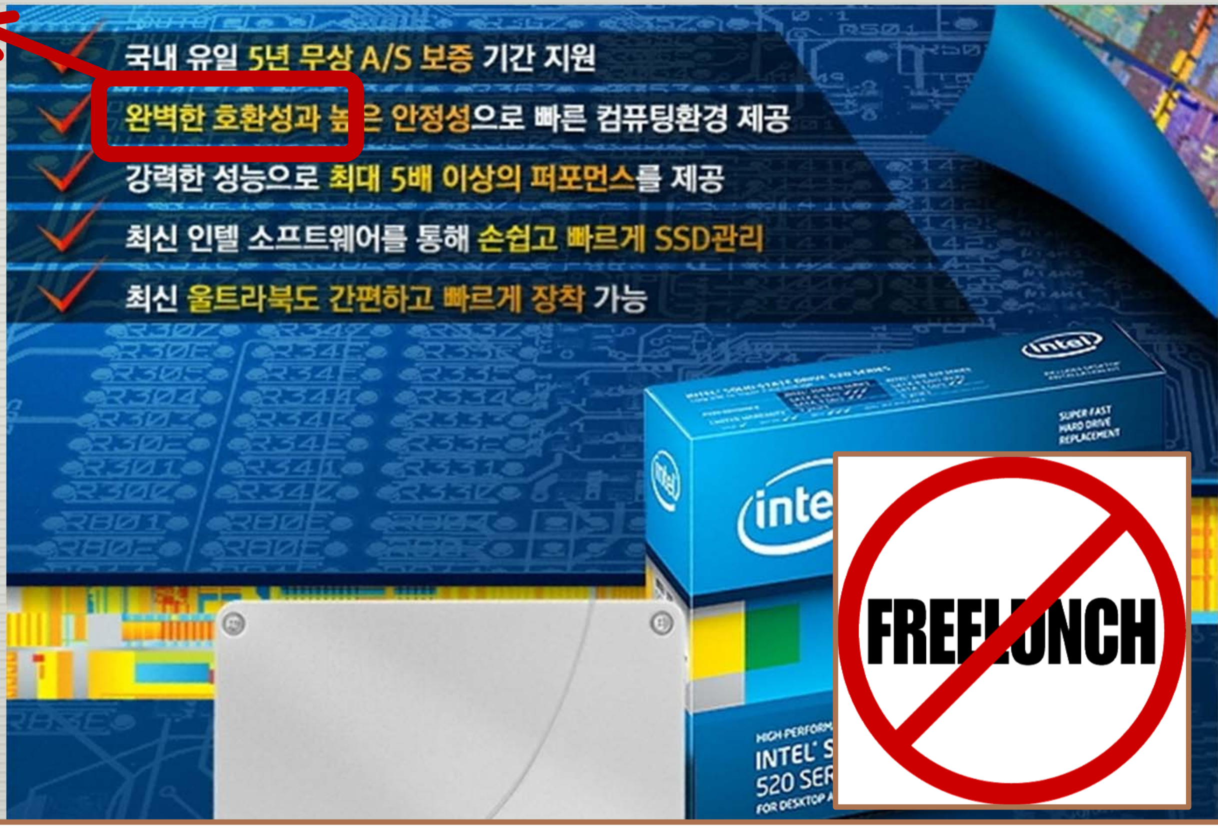
Introduction
- Ordinary Practice to Use SSD

No modification to Software

Application

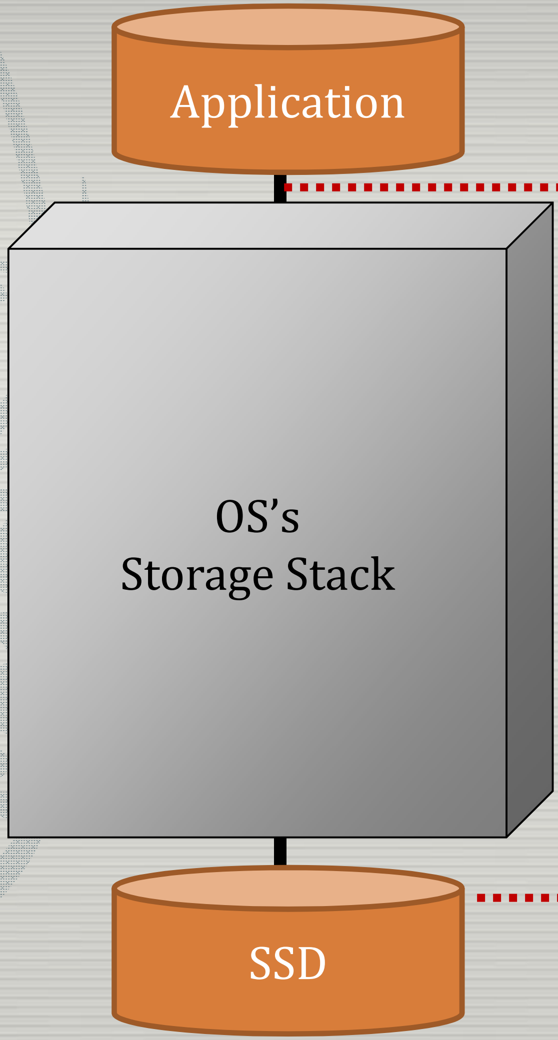
Operating System

Replacing the h/w only



No matter how fast storage devices get, Software consistently would eat up the extra speed !!

Motivation



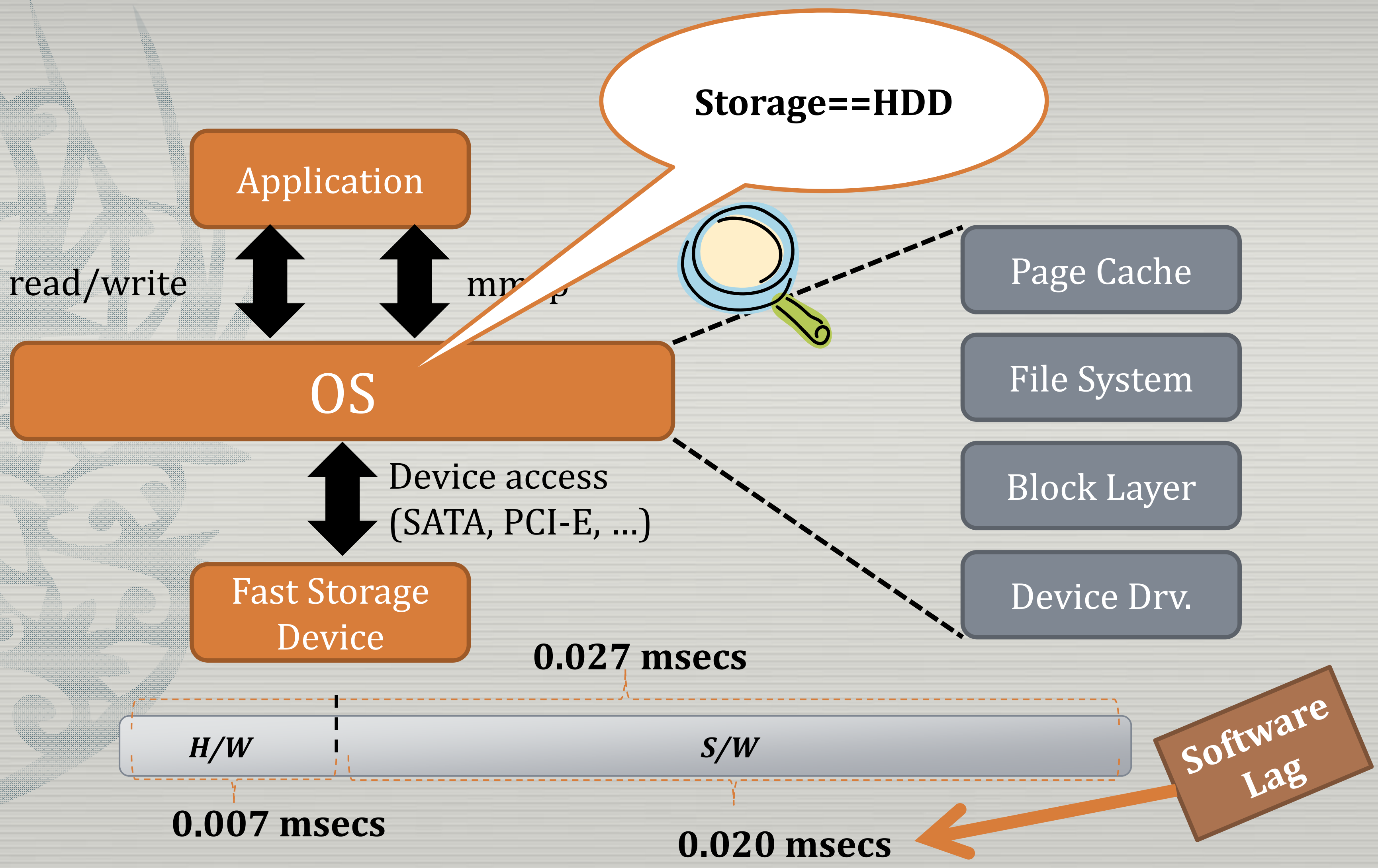
What matters us !
(Application throughput)

Different Performance Number!

What vendors give to us
(Device throughput)

e.g.) OCZ Vertex3(60GB, SATA3, EMLC)
➔Vendor's perf.: 60000~85000 (4KB IOPS)
➔Fio's perf: 10000~15000 (4KB IOPS)

The Problem
- OS is still in the Dark Age !



DRAM-SSD, 4KB I/O

OS Should be Re-Designed!

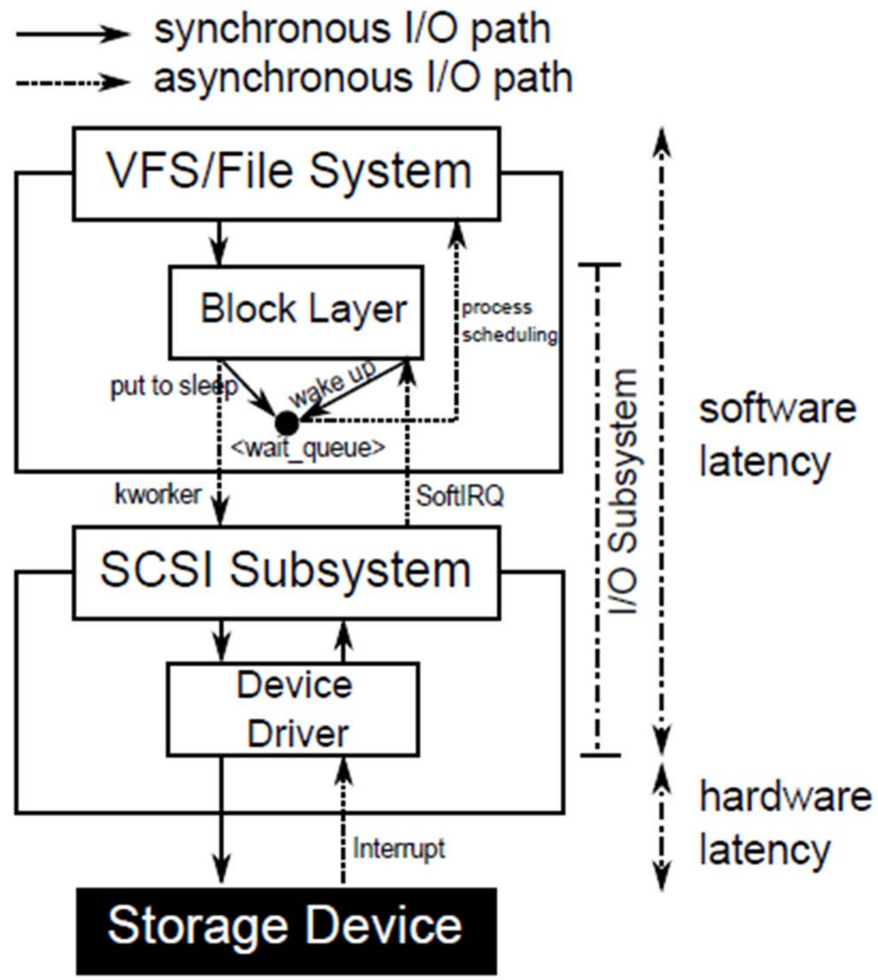


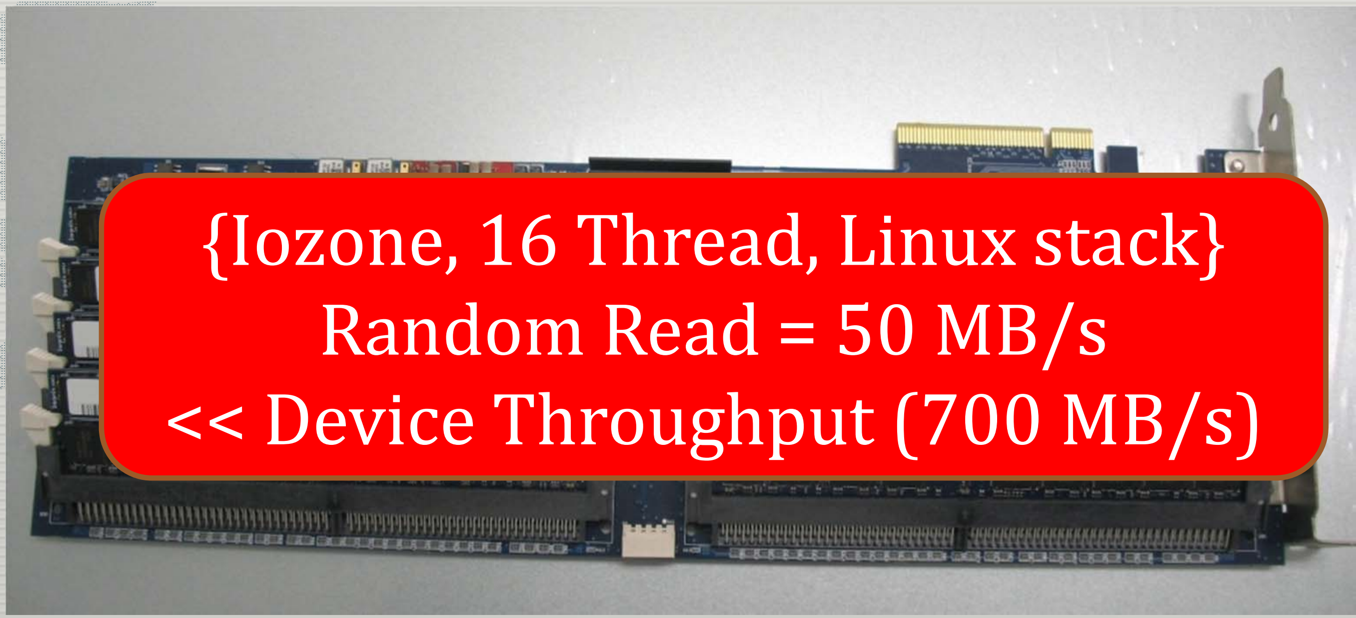
Figure 1: Common I/O path in Linux storage stack

[Source of Delays]

1. **Interrupt Overhead**
2. **Delayed Execution**
3. **Spatial Merge**
4. **Disk-Assumption in I/O scheduler**

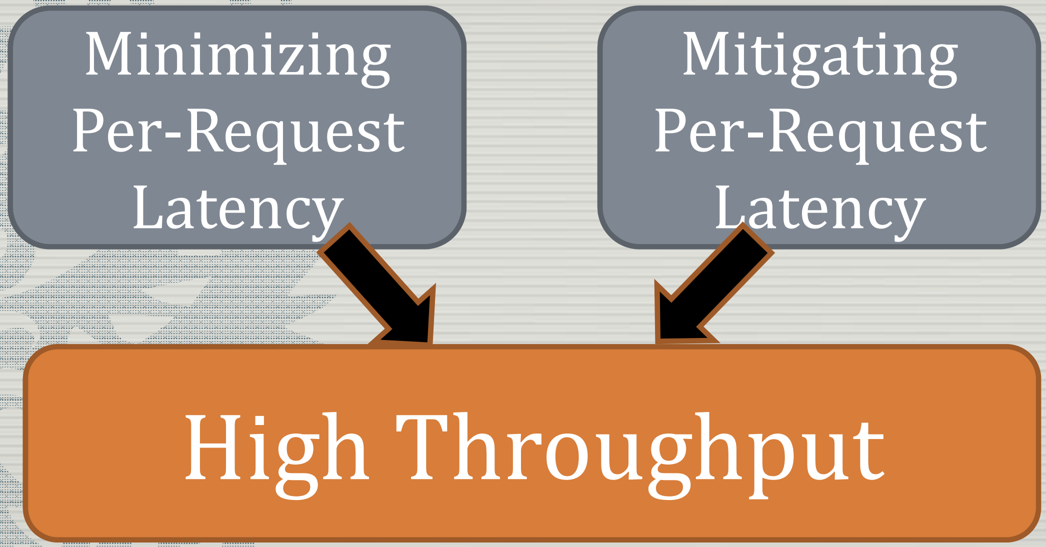
Our Experience

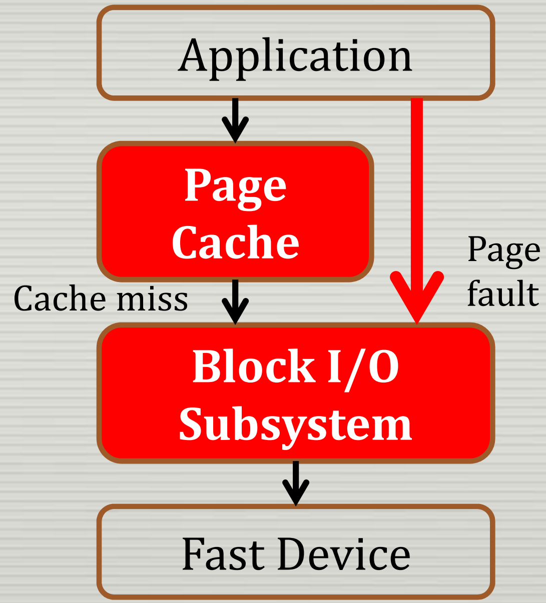
- **High-performance SSD**
  - DRAM-based SSD (provided by Taejin Infotech)
  - 7 usecs for reading/writing a 4 KB page
  - Peak device throughput: 700 MB/s ⟹ 1.4GB/s
  - DDR2 64 GB, PCI-Express type



{Iozone, 16 Thread, Linux stack}
Random Read = 50 MB/s
<< Device Throughput (700 MB/s)

Optimization Approach

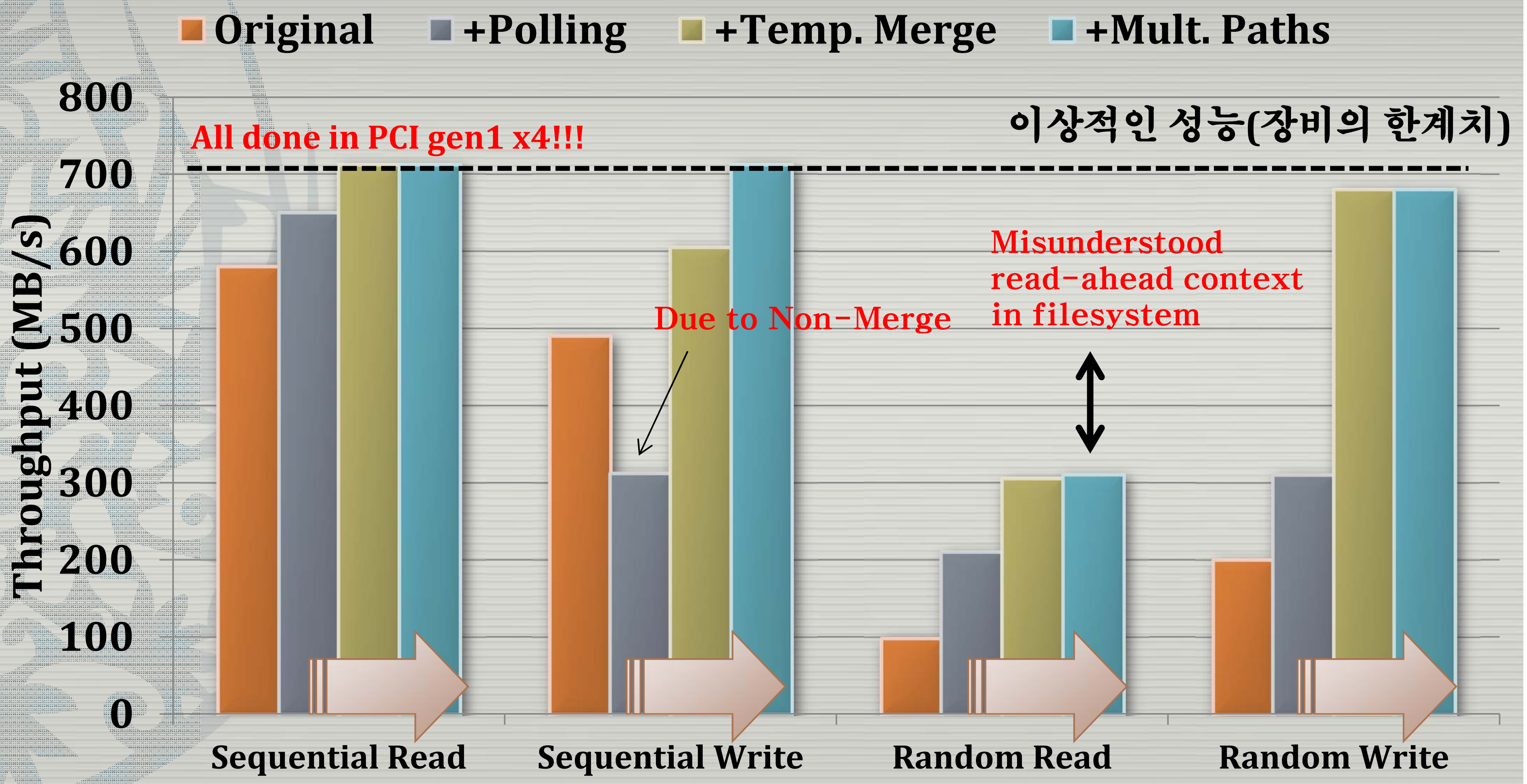




Optimization Techniques

- **Polling (vs Interrupt)**
  - Eliminate Asynchrony Delays
- **Temporal Merge (vs Front / Back merge)**
  - Maximize Merge Opportunity
    by Merging Non-Spatial Blocks
- **Multiple I/O Paths (vs Single I/O Path)**
  - Enhancing background
    single thread I/O merge opportunities

Before & After



Speedup	+23%	+45%	+215%	+240%

New Challenges

- Not much gain after H/W upgrades

>= 4KB requests

Approx **100%** IOPS Improvement

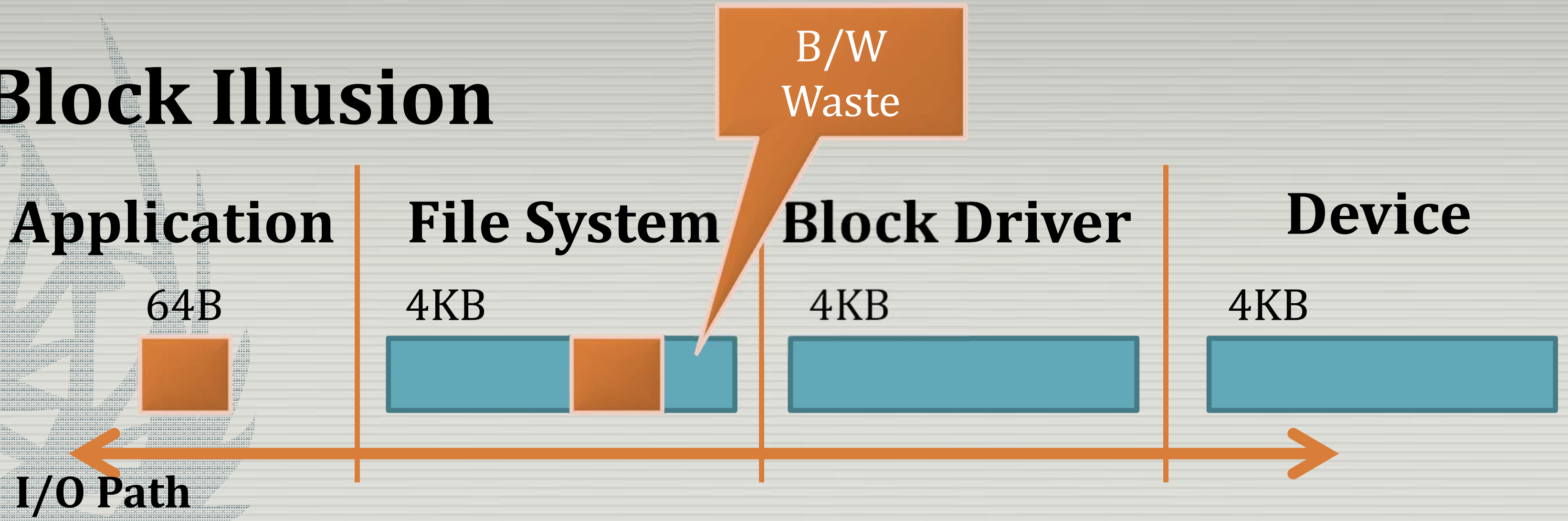
0.5KB requests

2x PCI-e Channel Bandwidth

Approx **25%** IOPS Improvement

New Challenges

- Block Illusion



- Preserving Latency while Batching I/O
  - Temporal Merge
  - Asynchronous Merge

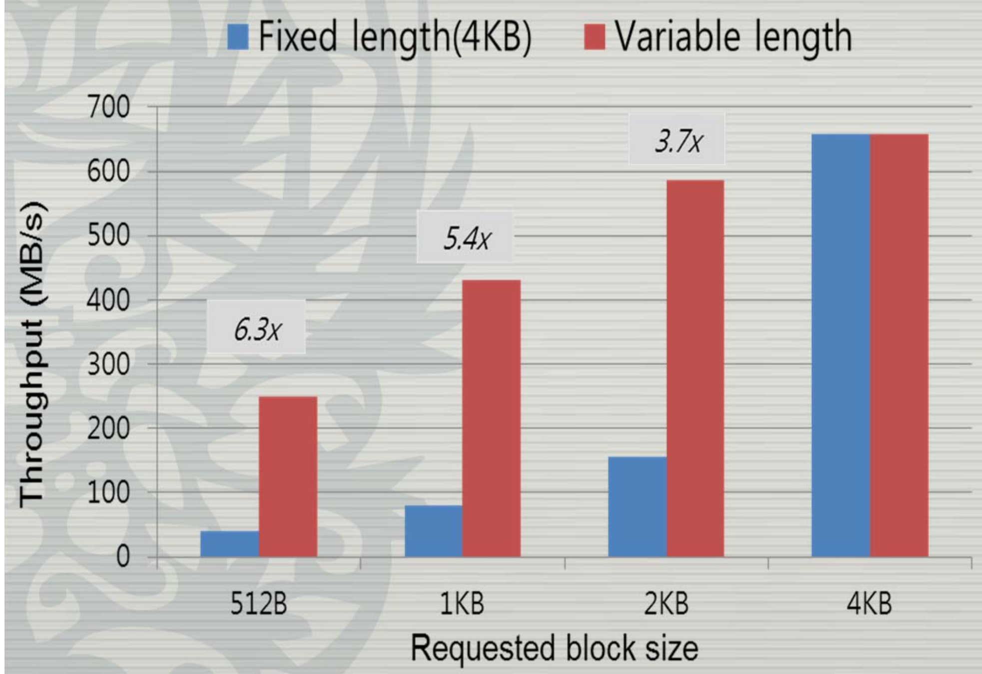
Addressing New Challenges

- **Sub-block/page I/O:**
  - Eliminate B/W waste

- **Pipelined I/O:**
  - Pertain per block I/O completion after merging

Addressing New Challenges

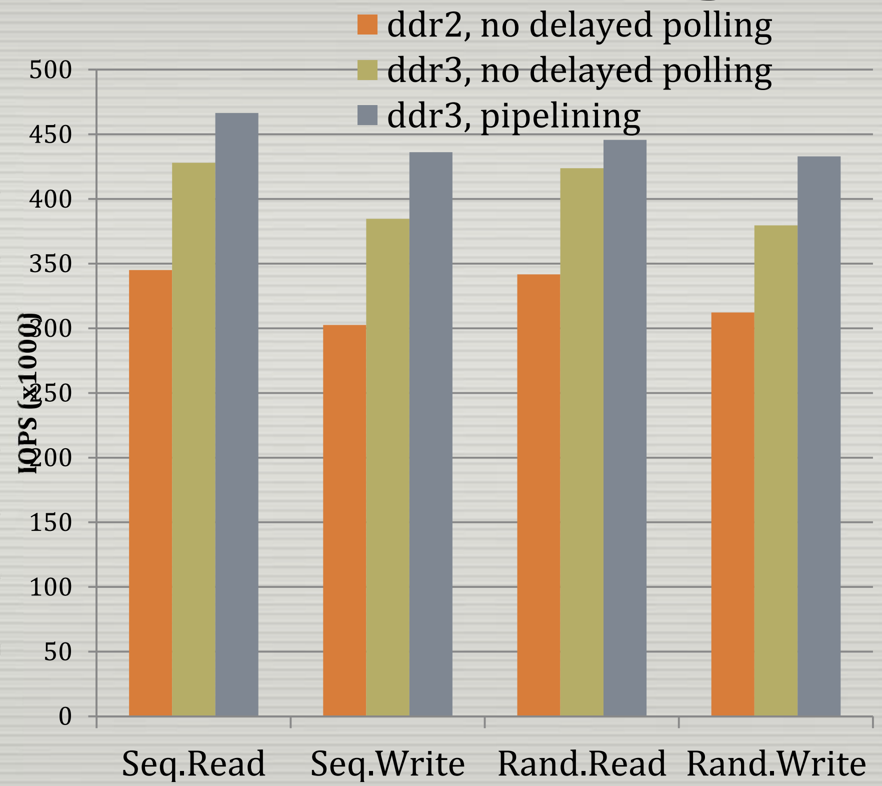
Sub-Block I/O



Sub Block I/O Improvement

3.7x ~ 6.3x 성능 향상.

I/O Pipelining



FIO, 512 byte, 16 threads

약 14% 성능 향상. (최대 60,000 IOPS 향상)

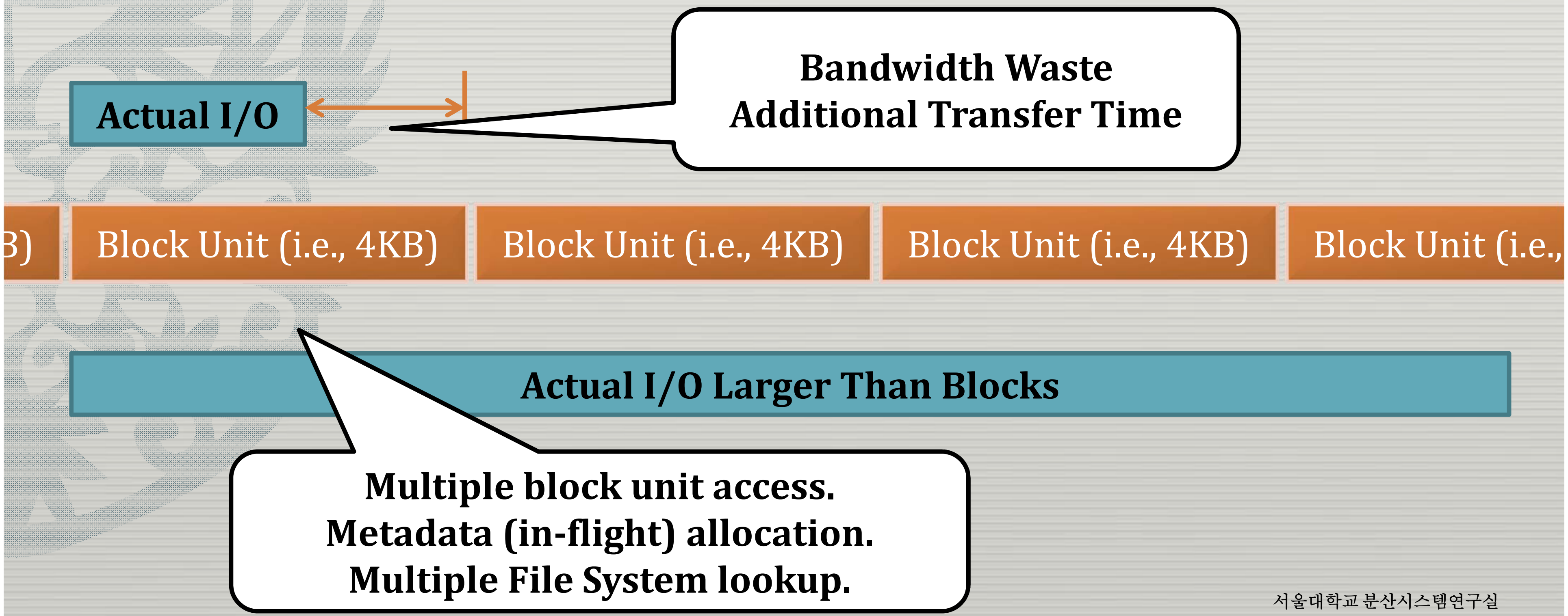
Are we good?

- **File System Overhead & Page/Buffer Cache Overhead**
  - Sub page I/O, Super page I/O all limited by block based File Systems & Page cache mgmt.
  - ***Need I/O interfaces not limited by "Blocks".***
- **No Standard Way for Direct Device Access**
  - Low latency design requires direct access to device queue & and tags.
  - OS Block / SCSI layer don't provide this.
  - ***Need a generic I/O interface to provide "Direct Access".***

Design 1 : Block-less IO

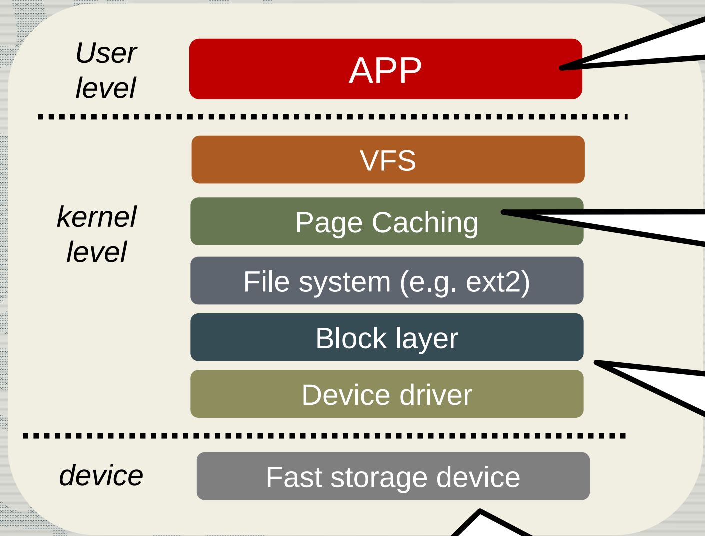
Byte Addressable I/O
"Block-less I/O Interface"

- **Block based H/W, S/W is a problem!**
  - Incurs B/W overhead
  - Incurs Additional Overhead



Can we Eliminate the Block I/F?

Block I/F based I/O stack.



Applications: Yes << No
Some apps can live without them.
Some cannot. *But most of them are optimized based on "blocks"*

VFS, Page Cache, File System: No!
Definitely based on blocks.
Optimization based on blocks.

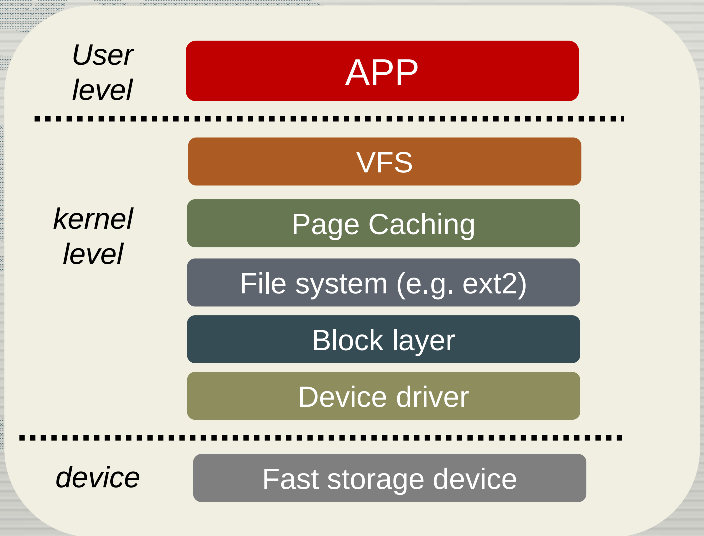
Block layer & Device driver: Yes!
Easy if the H/W supports it.
Just a matter of changing addressing schemes.

H/W: Yes
Can eliminate if vendor support is there.
"But they ask: Can S/W do that?"

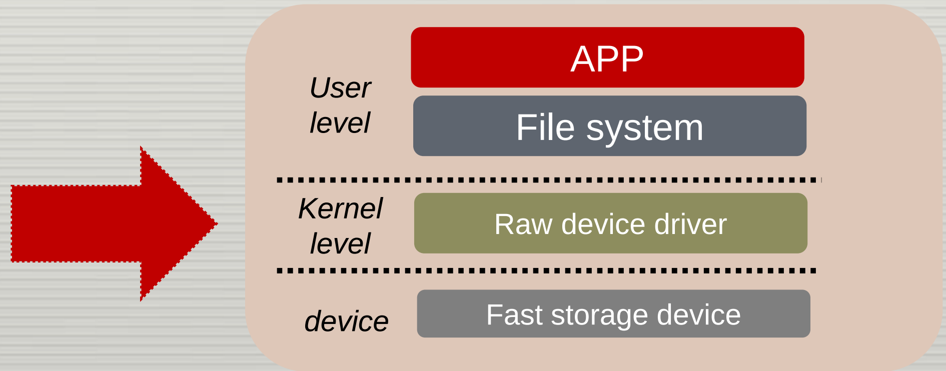
Block-less I/O Path

- **Re-design an I/O path free from blocks.**
  - Abandon page cache: Doesn’t help us much!
  - Fragment based FS: BAFS (Byte Addressable FS)
  - Byte addressable I/O driver & device
  - Synchronous I/O

Block I/F based I/O stack.



Byte Addressable I/O



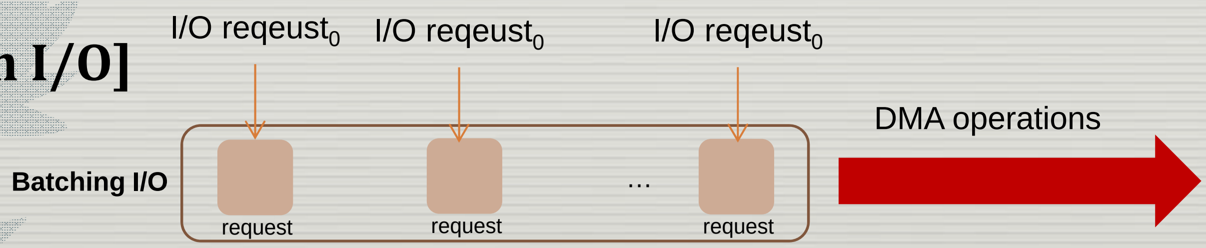
Block-less I/O Path: Overview

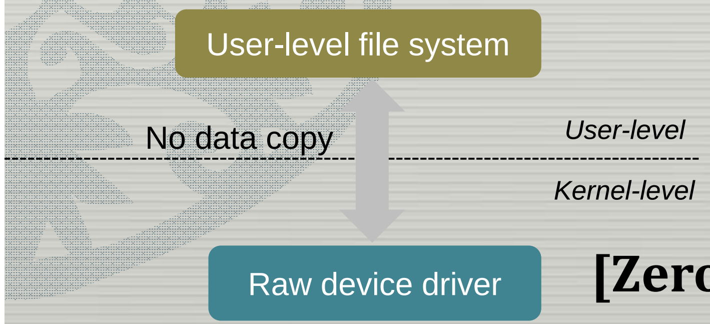
- **VFS, Page Cache-less**
  - Block based page cache management overhead.
  - Poor random I/O cache hit ratio.
  - Just additional overhead. → Let's live without it.
- **BAFS (Byte Addressable File System)**
  - Preserve I/O size as same as possible
  - Fragment Based instead of Block Based
- **Byte Addressable Raw I/O device driver.**
  - Proprietary Device Driver for Low Latency
  - Eliminate scheduler overheads
  - Other additional optimizations such as zero copy, batched I/O, swapping queues.

Byte Addressable Raw I/O device driver

- **Using I/O I/F not restricted to LBA's.**
  - Implemented as a character device.
  - <Command, Offset, Request size> based I/O
- **Internal Optimizations**

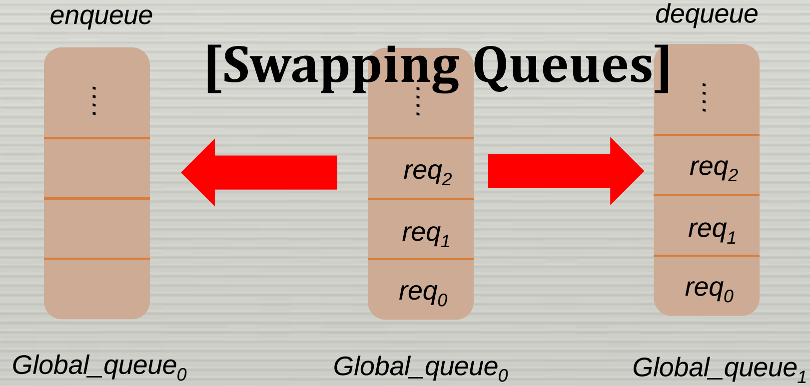
[Batch I/O]





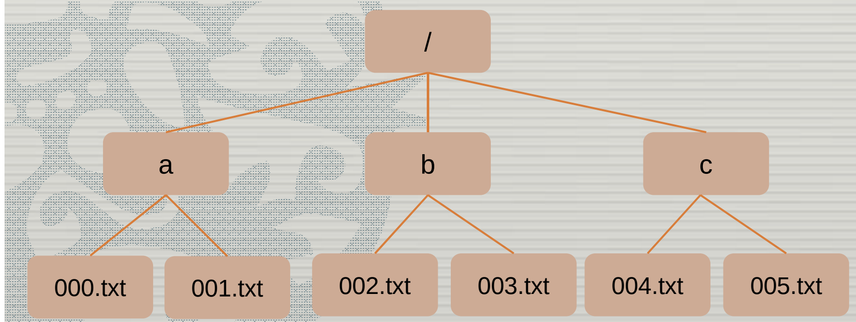
[Zero Copy]

[Swapping Queues]

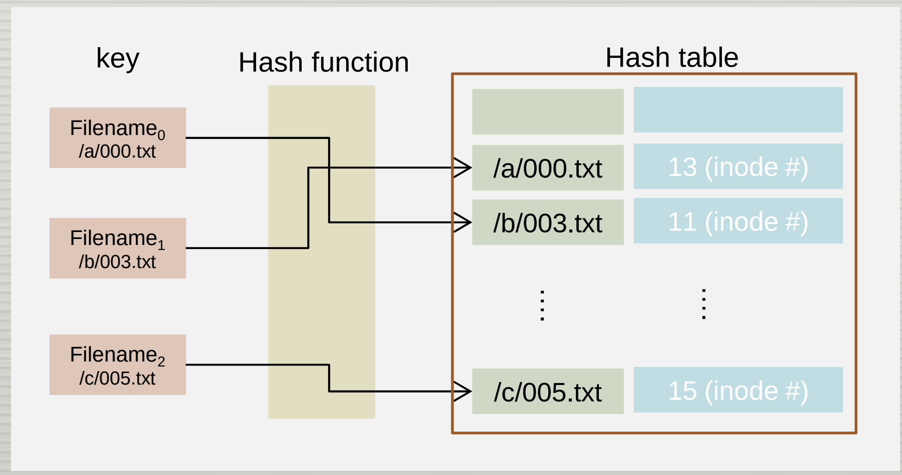


Byte Addressable File System

- **Double Hashing based Metadata Mgmt.**
  - Key: filename(including path) / Value: inode
  - Features flat directory access.



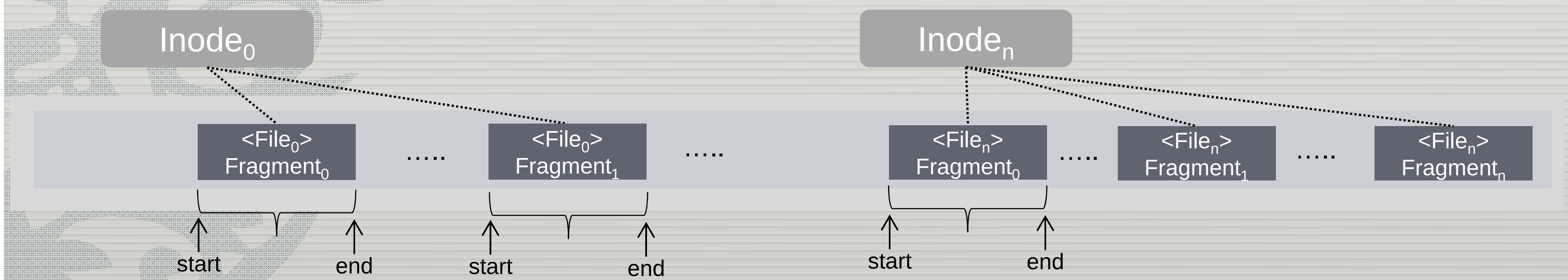
<Directory structure>



<Hash table for setting and finding inode>

Byte Addressable File System

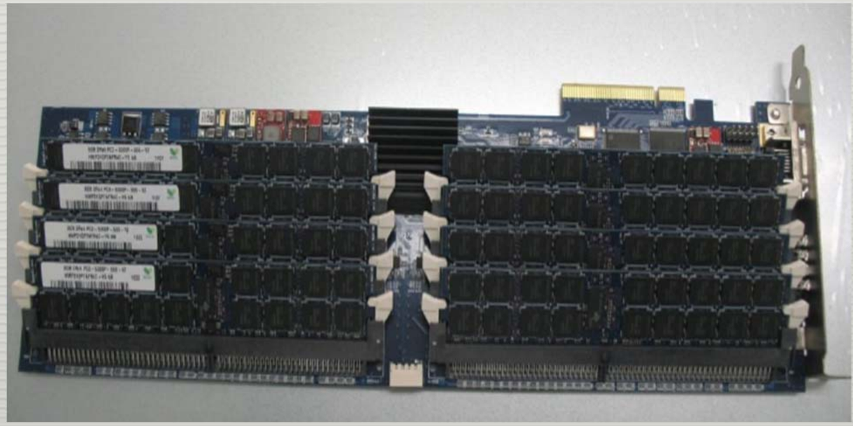
- **Managing data region**
  - A inode manages data region by using fragments in the B-tree
  - The fragments manitain information of data region
- **Fragments**
  - We use fragments for managing data region
  - It can manage the proper data region for user requested size



Target System

- **Hardware**
  - CPU: 8 cores (Intel Xeon E5630 2.5GHz)
  - RAM: 8GB
  - DRAM-SSD
    - Peak device throughput: 700~750MB/s
    - DDR2 64GB, PCI-Express type

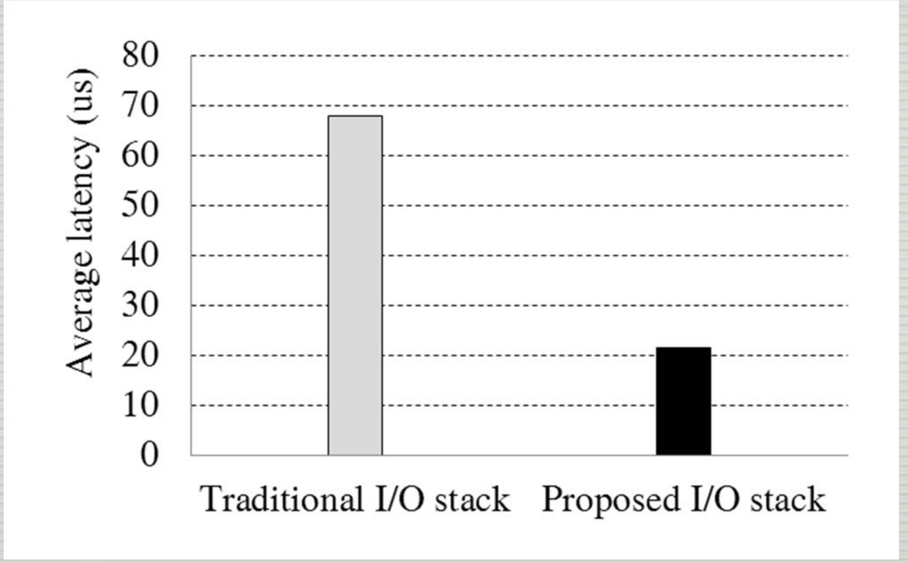
- **Software**
  - Linux kernel 2.6.32
  - FIO benchmark
  - IOZONE benchmark



Latency Reduction

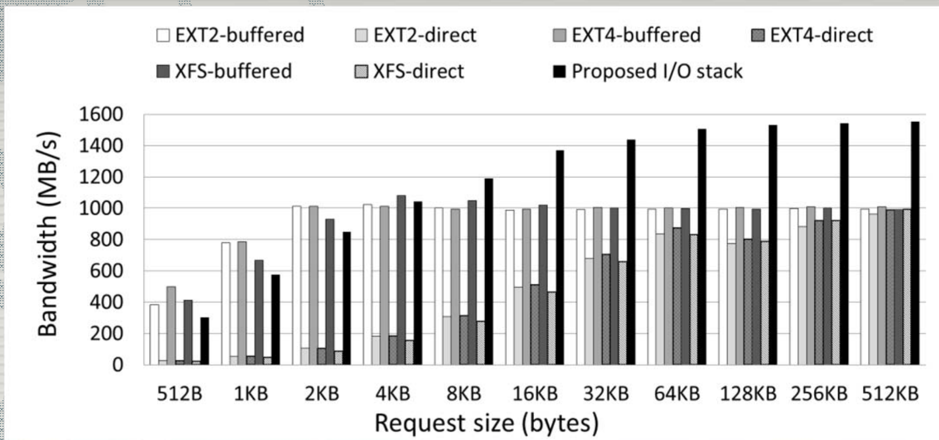
Layer	Functions	In/Out	Time (us)
VFS	sys_read	in	0
	do_sync_read	in	0.5
FS	ext2_readpage	in	4.5
BLK	generic_make_request	in	8.5
	generic_make_request	out	11.5
SCSI	scsi_request_fn	in	13.5
	scsi_request_fn	out	20.5
BLK	io_schedule	in	22.5
DEV	SSD_intr	in	HL+22.5
	SSD_intr	out	HL+31.5
BLK	blk_done_softirq	in	HL+40.5
	bio_endio	in	HL+43.5
	bio_endio	out	HL+45.5
SCSI	scsi_run_queue	in	HL+49.5
	scsi_run_queue	out	HL+50.5
BLK	blk_done_softirq	out	HL+51.5
	io_schedule	out	HL+55.5
FS	ext2_readpage	out	HL+57.5
VFS	do_sync_read	out	HL+60.5
	sys_read	out	HL+60.8
Total time : 67.8us			

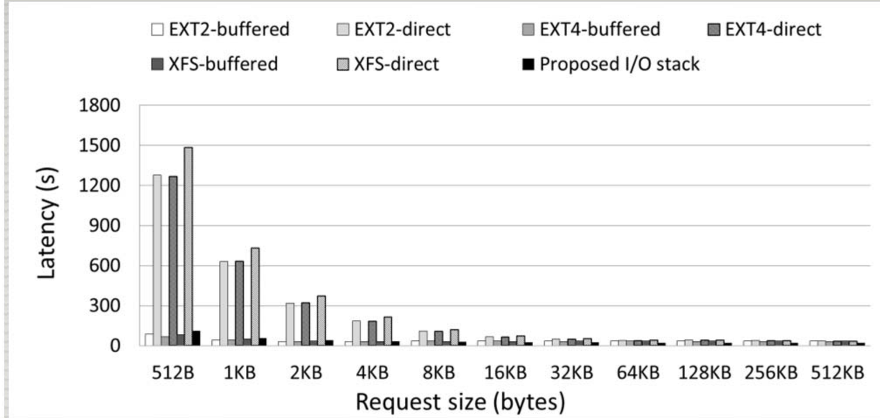
The reduction for I/O latency by about 3 times



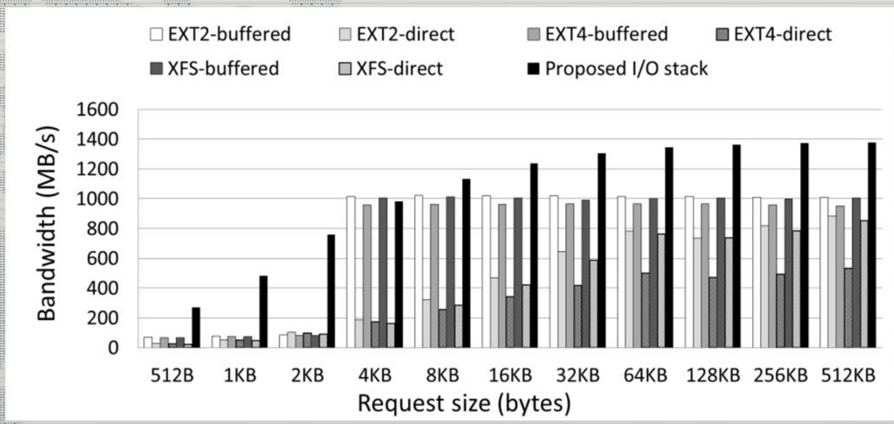
Experiment Results (FIO – DDR3)

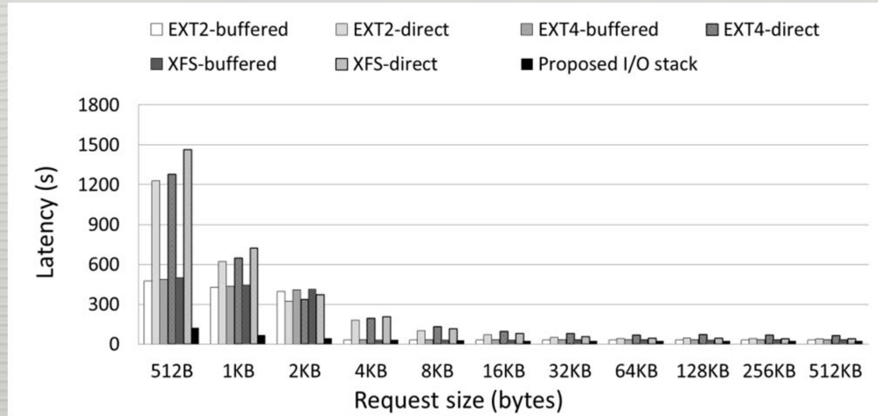
Sequential Read – I/O bandwidth and latency





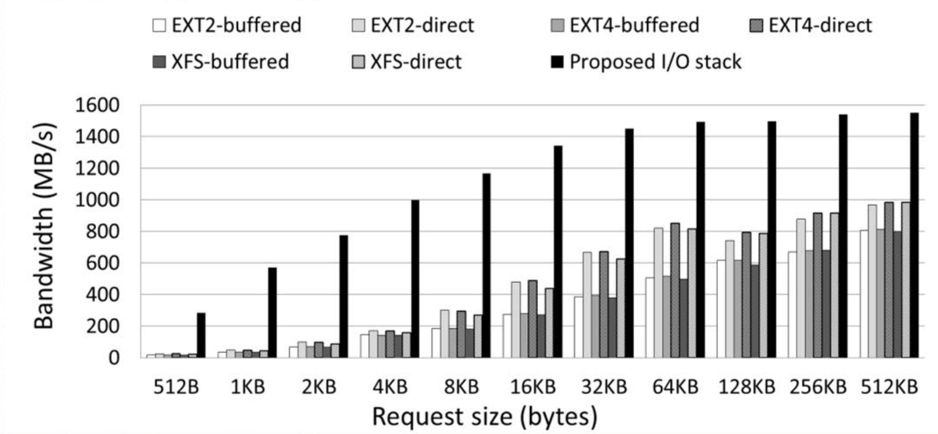
Sequential Write– I/O bandwidth and latency

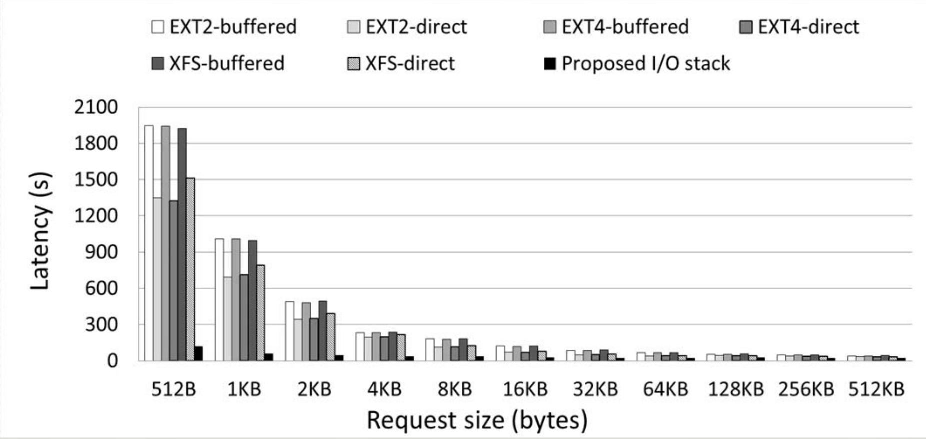




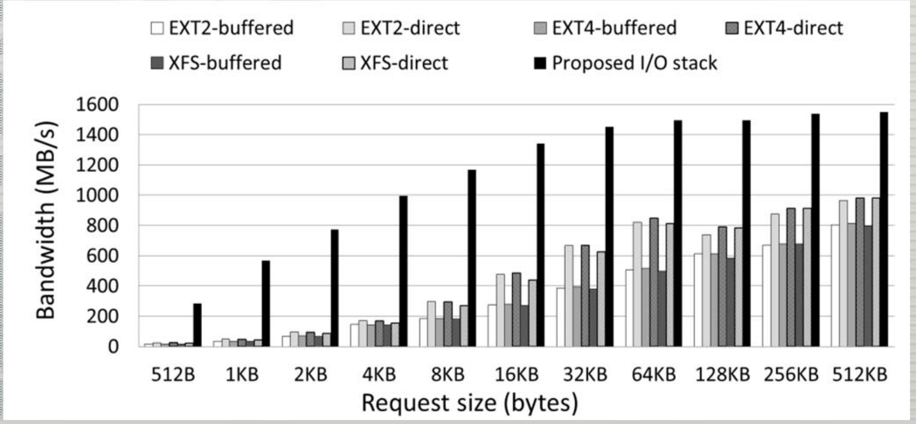
Experiment Results (FIO – DDR3)

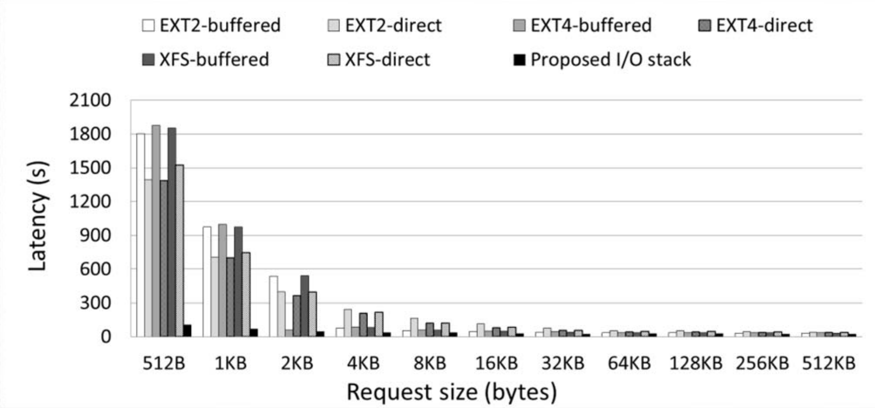
Random Read – I/O bandwidth and latency





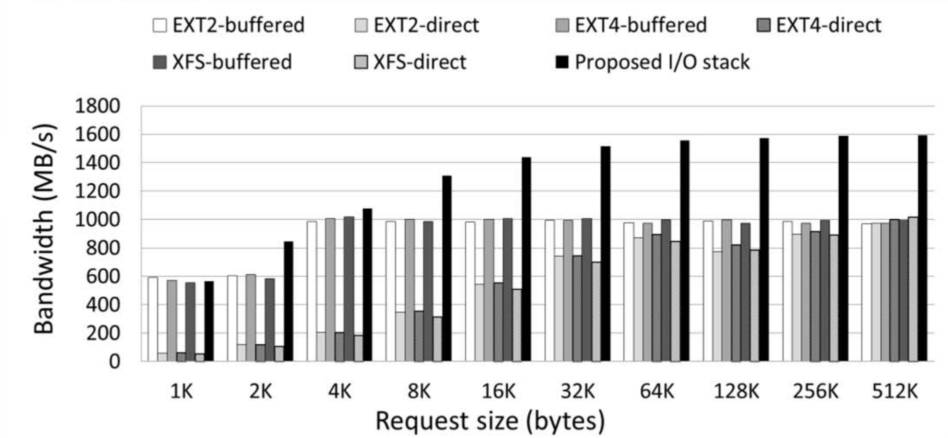
Random Write– I/O bandwidth and latency

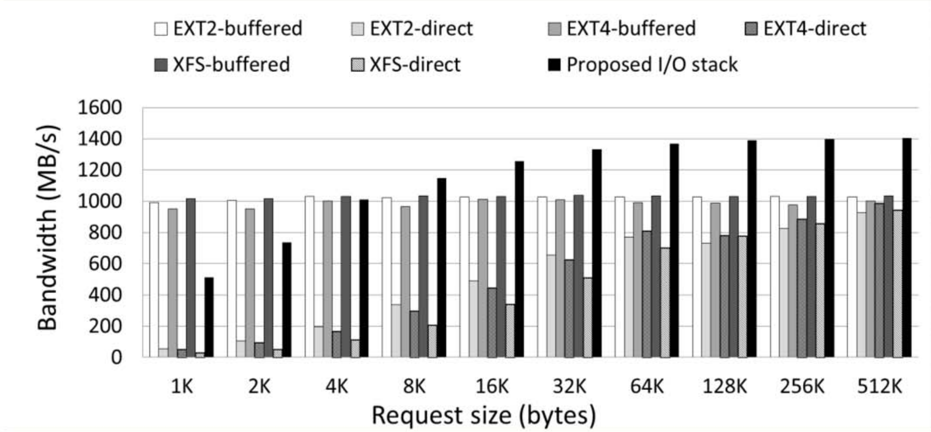




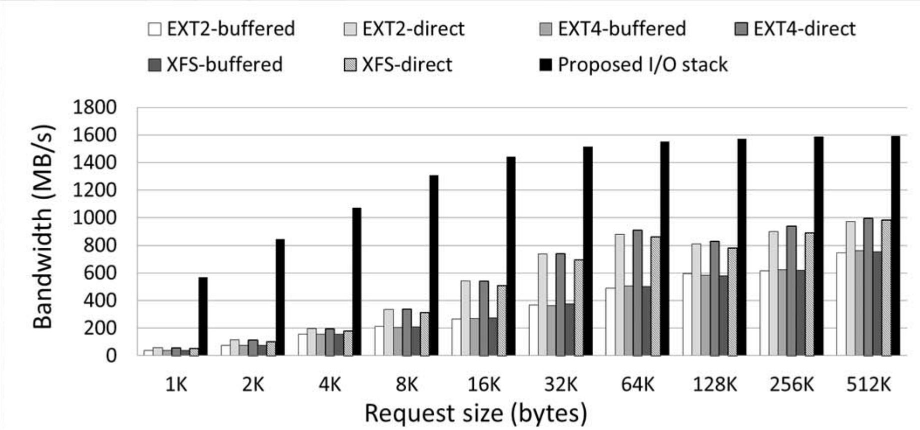
Experiment Results (IOZONE – DDR3)

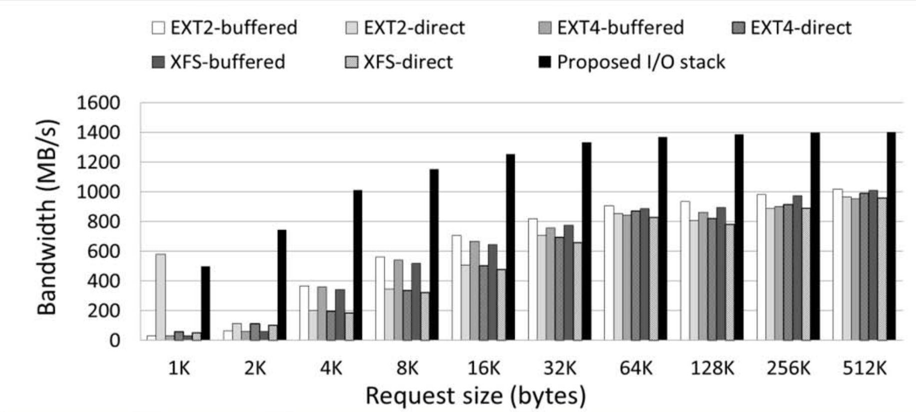
Sequential Read and Write– I/O bandwidth





Random Read and Write– I/O bandwidth





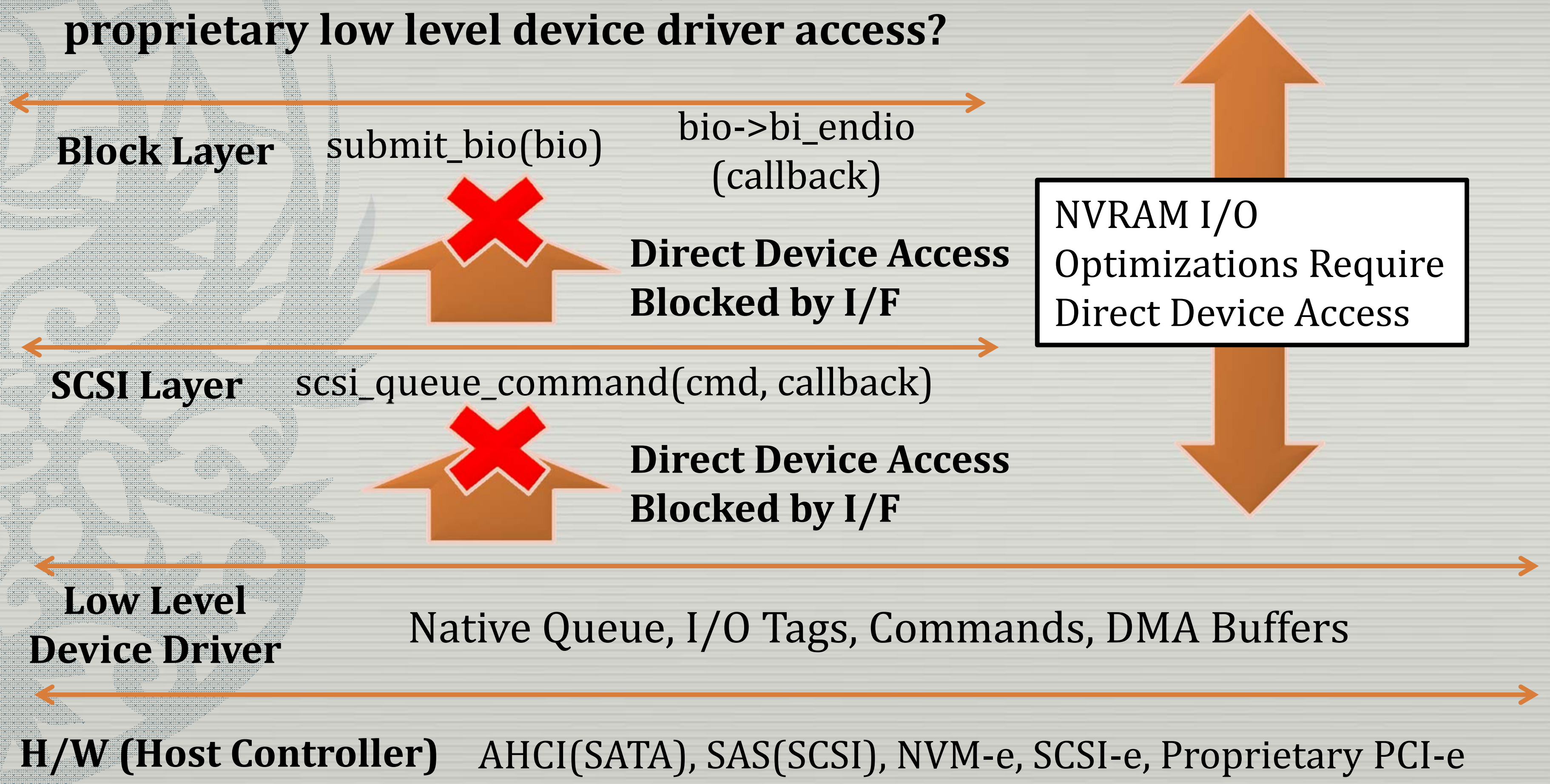
Design 2 : Enhanced Block IO

Embracing Block Based Eco-system

- **Synchronous I/O is not the only I/O path**
  - We have buffered I/O, mmap I/O,
    page swap I/O, Asyncronous I/O (aio)…
- **We can't live without blocks…**
  - **Userespace:** So many previous applications are based on block I/O.
  - **Kernelspace:** There are more OS S/W based on blocks or even chunks. (RAID, Volume mgr & etc…)
- **Standardized Block Based I/O Controllers**
  - Block based controllers & devces are already dominant.
  - But they also experience S/W limitations…

Current Device Abstraction is Not Enough – Too Coarse!!

i.e. Can we implement polling without proprietary low level device driver access?



We Need a Standard I/F for NVRAM Storage I/O

- **queue->make_request() is not enough!**
  - Redundant proprietary device driver code infecting upper layers. → Just a workaround!
  - Monolithic I/O strategy + Device Driver code.
- **VFS, Page cache, FileSystem, User App Access.**
  - I/O optimization is not restricted to block I/O layer & device drivers.
  - A standard way for the upper layers to ***<u>'see'</u>*** the device is necessary.
  - Standard I/F to provide a way to ***<u>'standardize'</u>*** I/O optimizations to the rest of the OS.

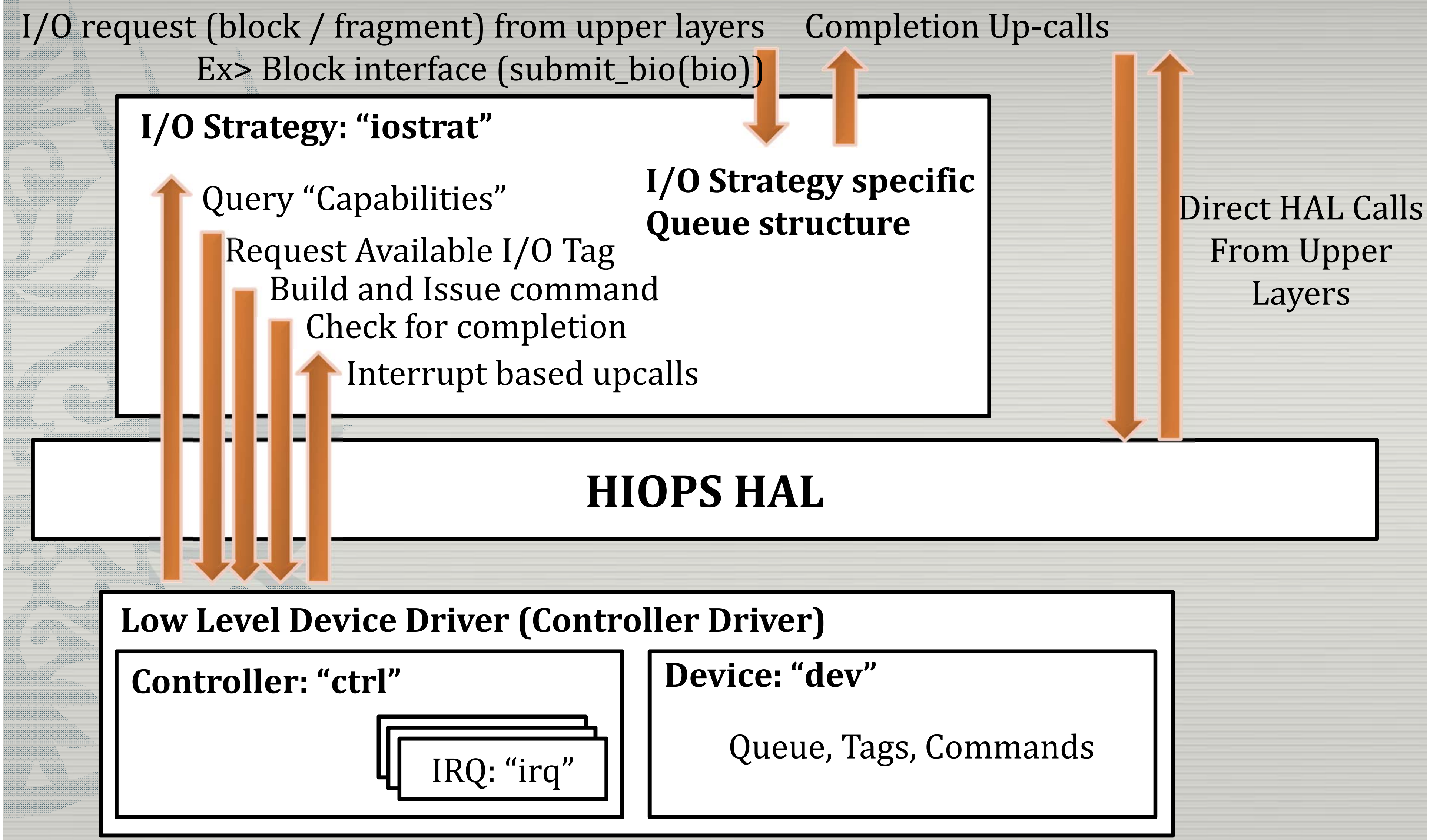
Opposite Approach: Block-full I/O

- **HIOPS-Hardware Abstract Layer**
  - Expand NVRAM optimizations beyond the block I/O layer.
  - Provides NVRAM H/W Low Latency Direct Access to upper layers.
- **Expand the use of H/W direct access API**
  - We can apply various optimizations based on these Low Latency Direct Access Operations.
  - Apply these optimizations to the upper layers. (i.e. page cache, swap I/O, RAID, userspace API)

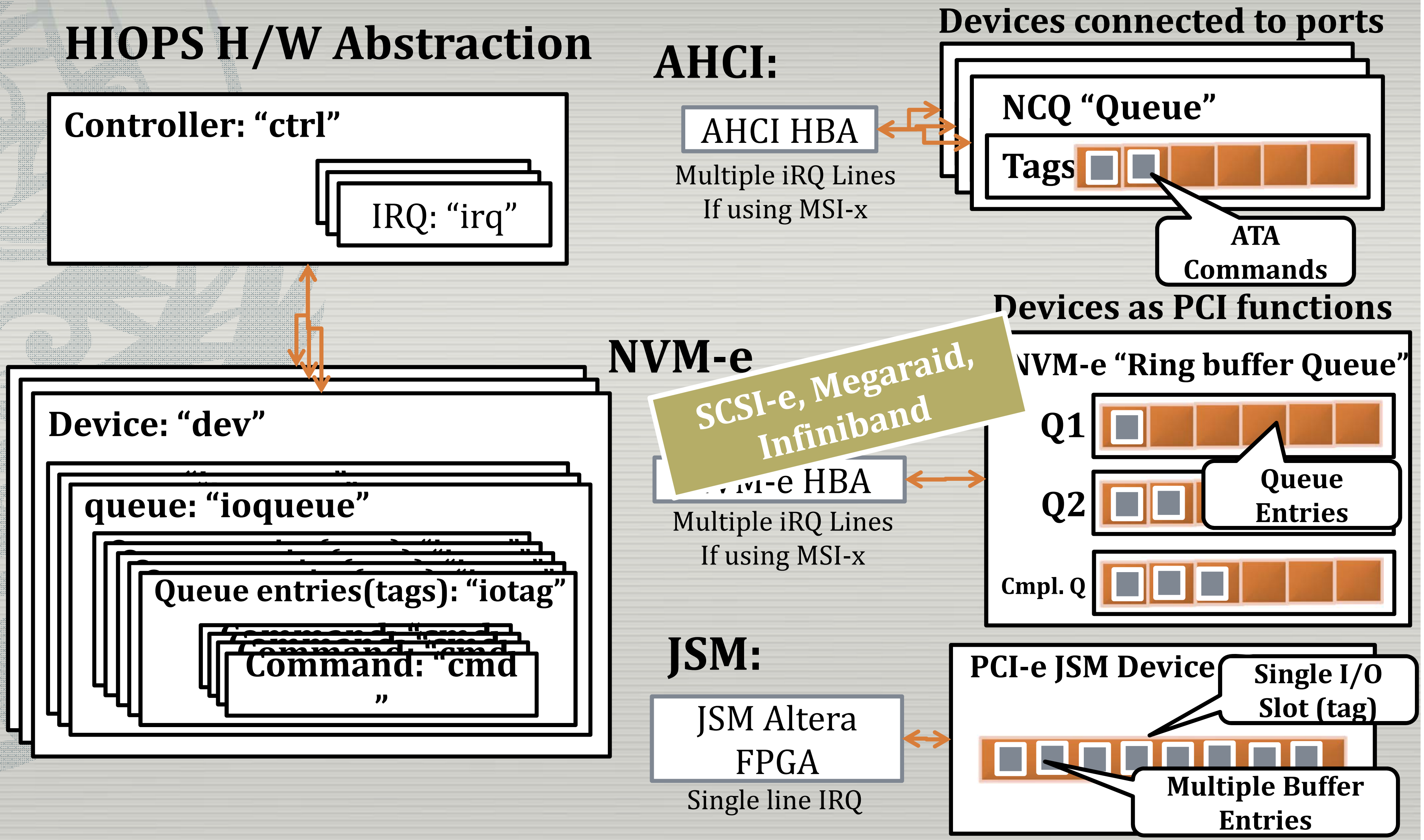
HIOPS Hardware Abstract Layer

- "Fine Grained Device Abstractions"
  - Buffers, Commands, Queues, Tags and Corresponding Operations.
- Define 'direct' operations to these device abstractions.
  - i.e. Is I/O on tag pending?
  - i.e. Do we have free I/O slots(tags)?
  - i.e. Map a command or commands to a tag
- Serves as a boundary for device driver issues.
  - Isolates software issues from generic OS parts.
- No overhead: Just a function pointer call.
  - i.e. VFS layer implementation

HIOPS Hardware Abstraction Layer

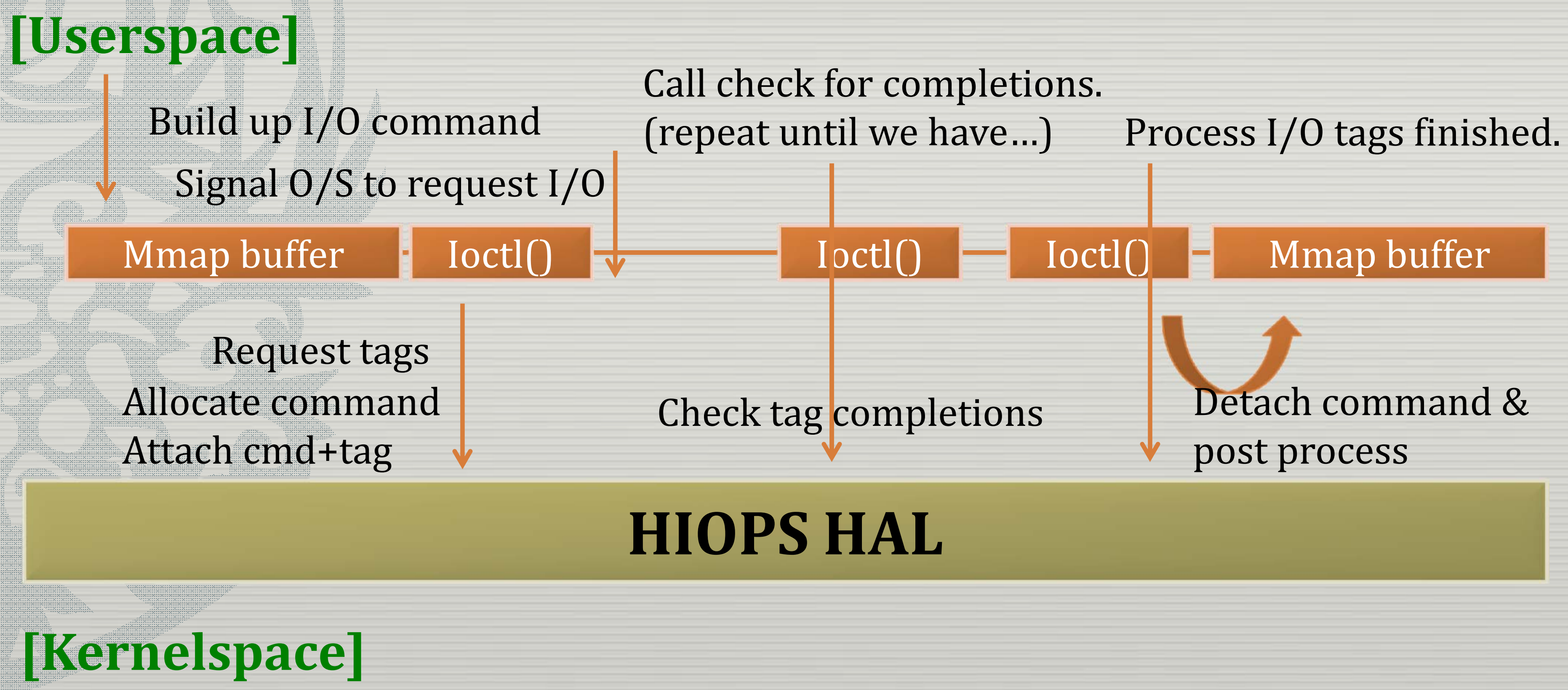


Host Controller-wise Application of HIOPS HAL



Case Study 1: User Level Polling

- A user level application polling based synchronous I/O (An extreme case)



Case Study 2: Page cache I/O batch

- **Write buffer I/O incurs small I/O problem. How about batching them in one go?**
  - Batch dirty blocks on one device round trip…

Page cache flush thread flushing dirty pages.

Flush thread collects dirty pages.

Request multiple I/O tags

Allocate multiple command
Attach multiple cmd+tag

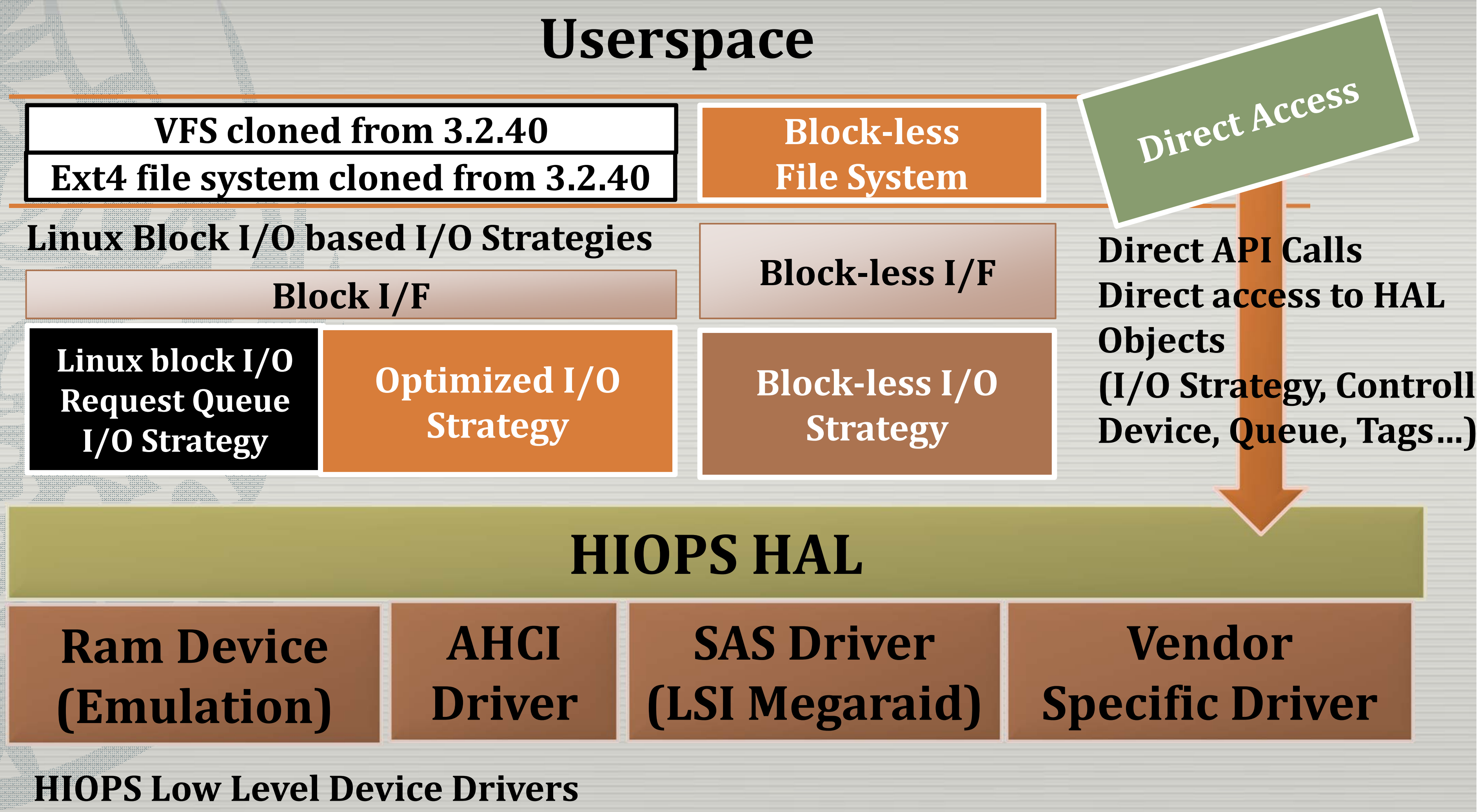
I/O Device Context

Check tag completions

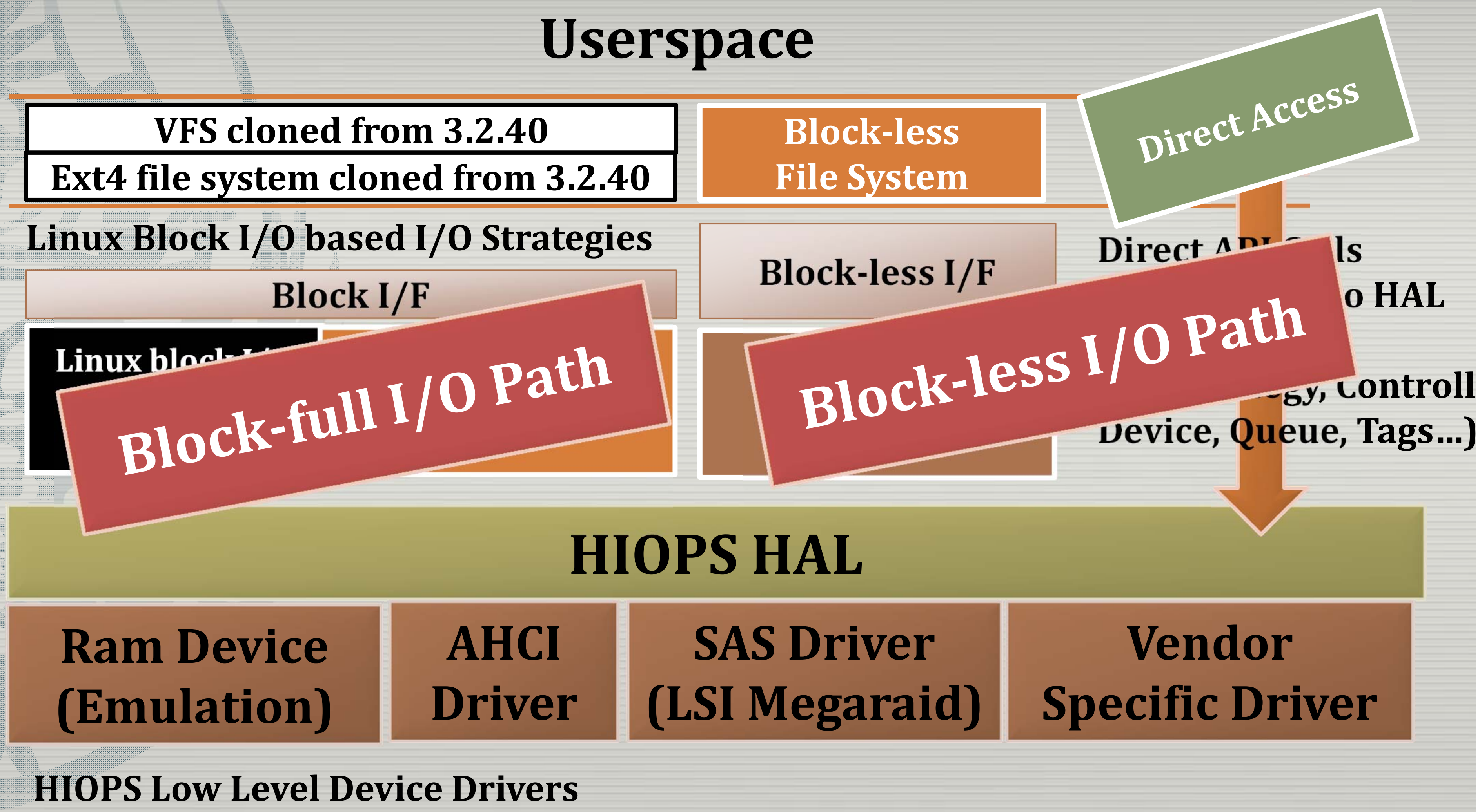
Detach command &
post process

HIOPS HAL

HIOPS-HAL based I/O stack



HIOPS-HAL based I/O stack



Conclusion

- **Every element of OS should be revisited if an application wants to benefit from fast storage devices.**
  - Our experiences prove it.
    - Block I/O subsystem, VM subsystem (mmio, page cache), Networked storage stack
- **Faster devices require changes to how we do I/O. New S/W, H/W Interfaces should be considered.**
  - Block-less I/O
  - Block-full I/O