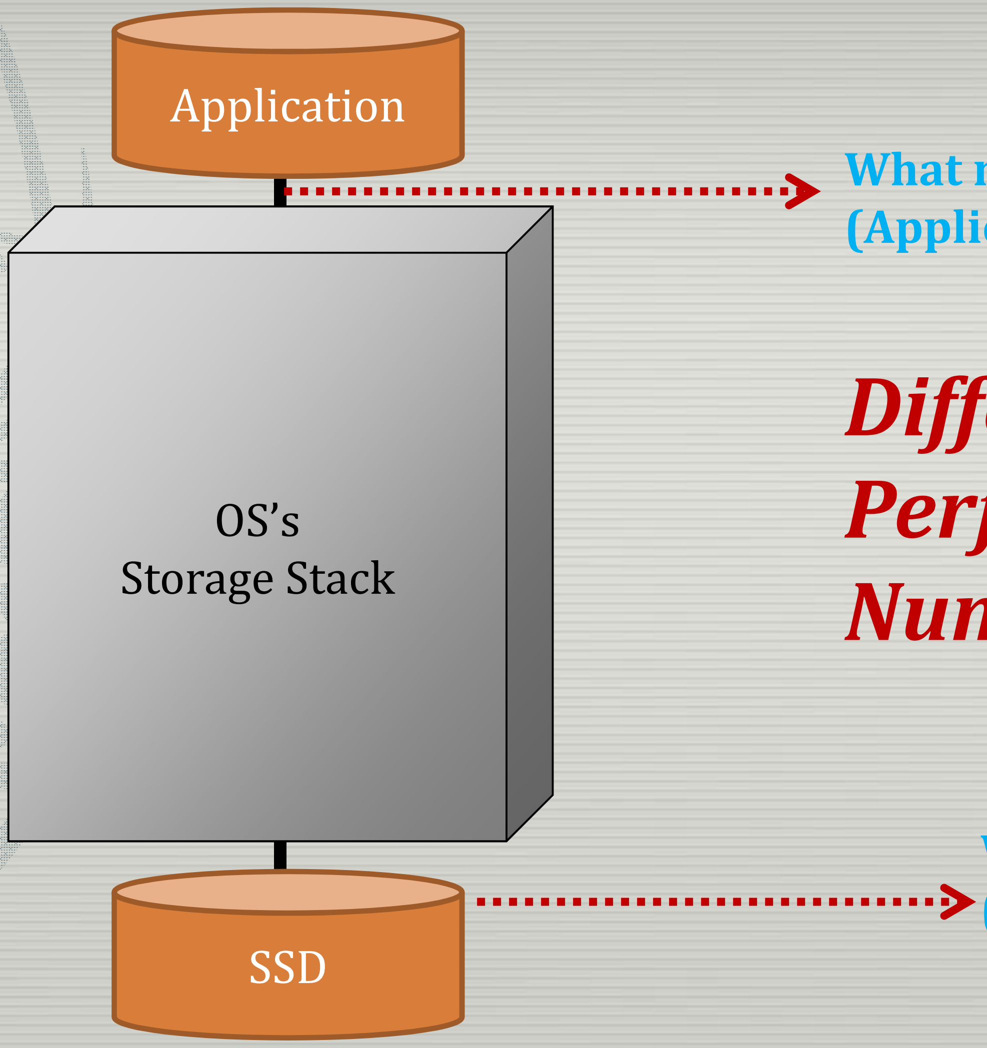
Achieving Peak Device Throughput for Random IO Workload

Dong In Shin
Taejin Infotech

Introduction

- I/O demand is very high.
  - Social Network Services
  - Cloud Platform
  - Desktop users
- Storage system has suffered from small random I/O accesses
  - Random throughput of a disk < 1 MB/s
- Fast Next-generation storage devices are coming.
  - Access Mechanism: Magnetics → Electronics
    - Low-latency → Good for random I/O performance
    - Flash-SSD, DRAM-SSD, PCM-SSD, …

No Free (Performance) Lunch

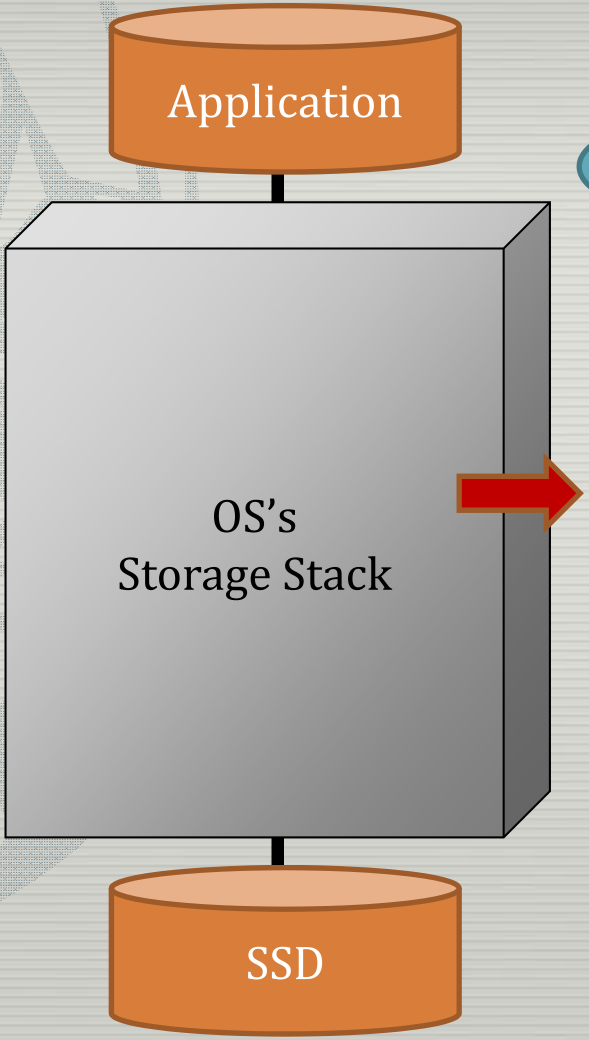


What matters us !
(Application throughput)

Different Performance Number!

What vendors give to us
(Device throughput)

Common Optimization



MSST'10, "High Perf. SSD ...."
MICRO'10, "Moneta: ..."
HotStorage'11, "Onyx: ..."
FAST'12, "When Poll is better ..."

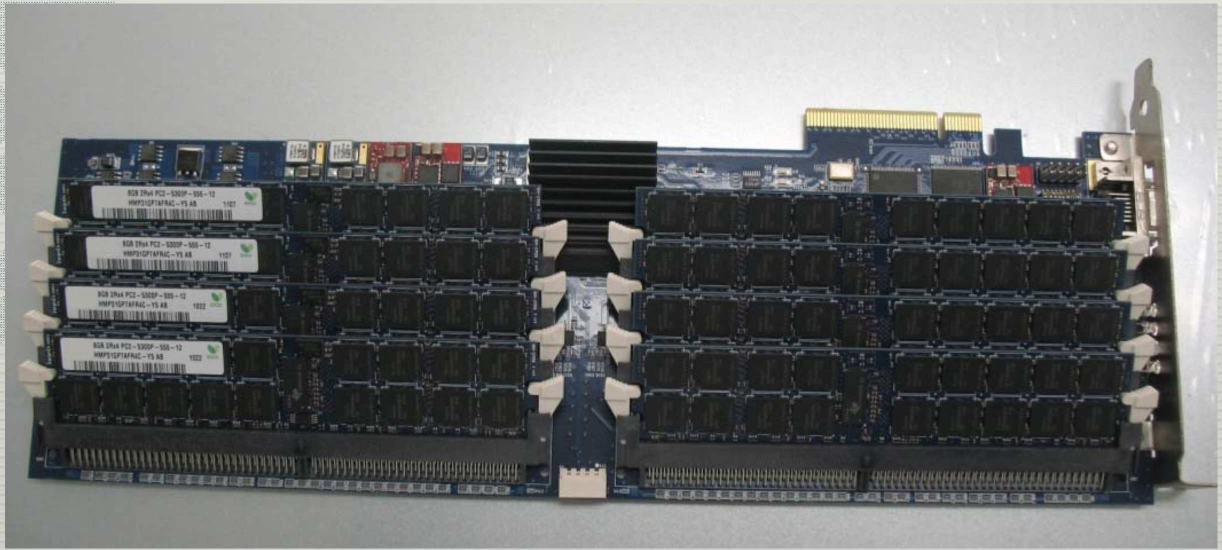
Synchronous I/O Path

1) Use Poll instead of Interrupt
2) Remove Delayed-Execution
   (e.g. I/O scheduler, SoftIRQ handler)

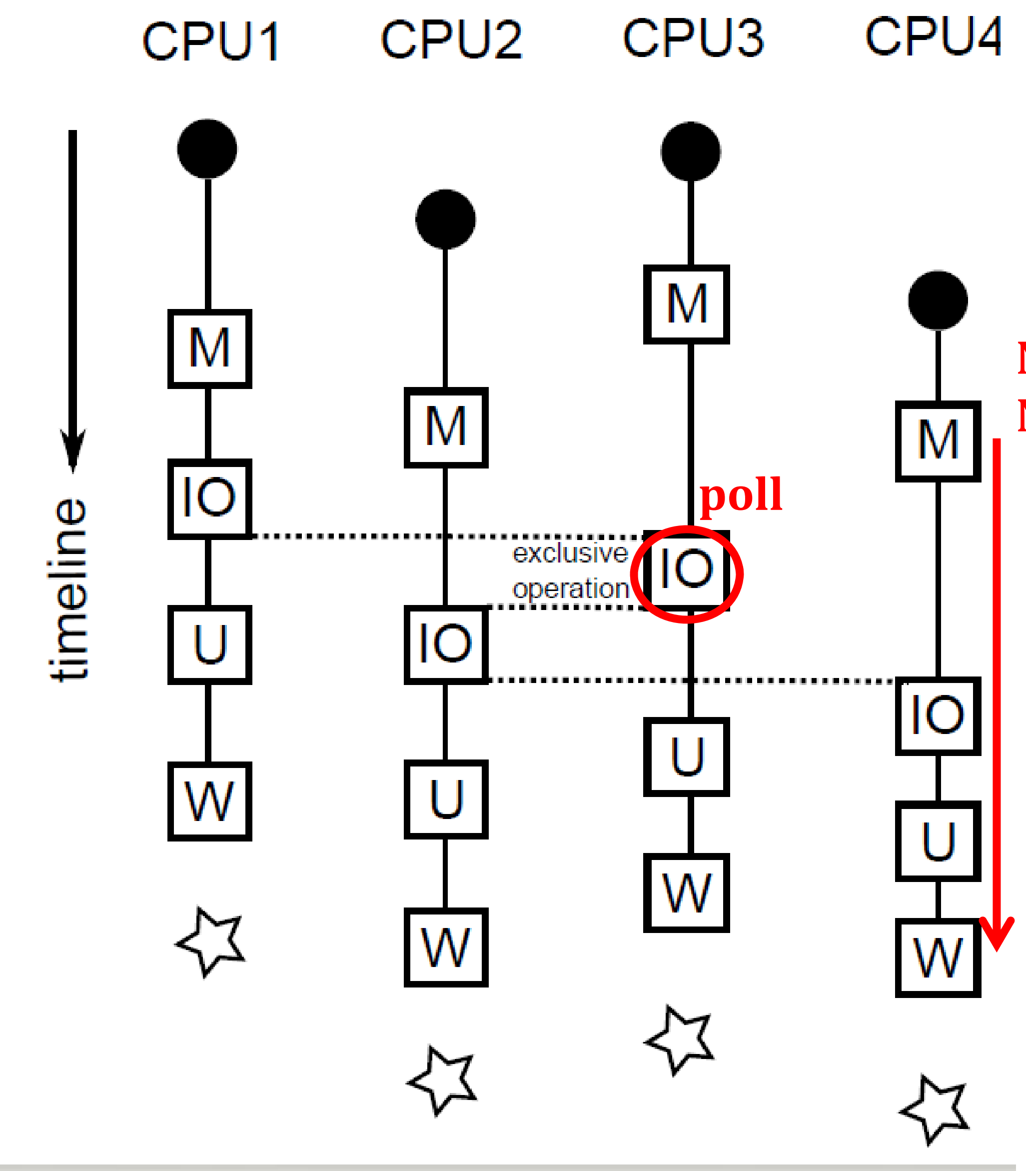
We will call it "Sync+Poll"

Evaluation of (Sync+Poll)

- Jetspeed DRAM-SSD
  - Next generation SSD developed by TAEJIN Infotech.
  - DDR2 64 GB, PCI-Express interface.
  - 7~8 usec for reading/writing a 4KB page
  - Peak device throughput: 700 MB/s



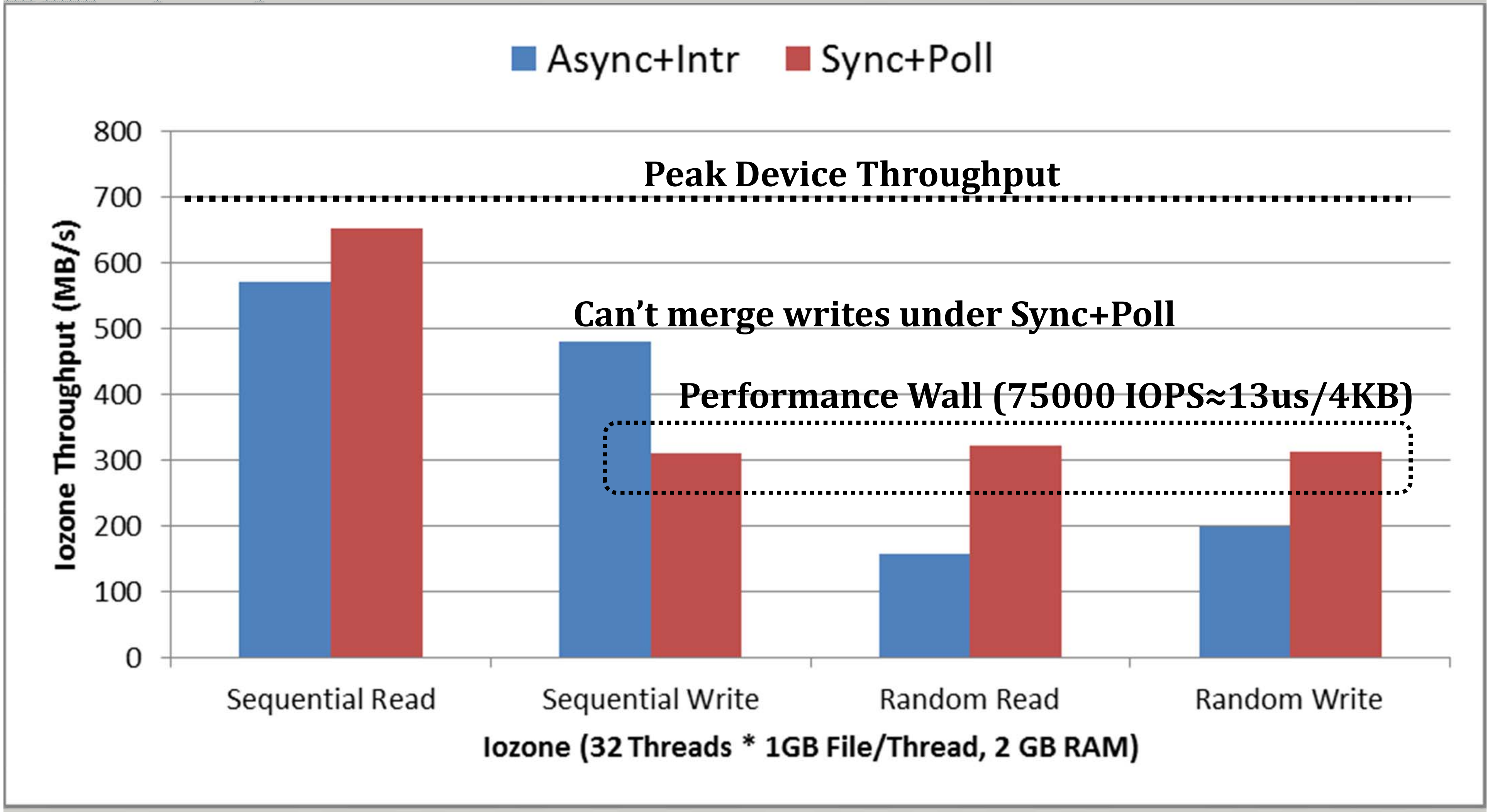
Evaluation of (Sync+Poll)



No “kblockd”
No “SoftIRQ”

Symbol	Description
●	The current I/O request
☆	The next I/O request
M	DMA pre-processing (Map)
IO	Start I/O and poll the completion
U	DMA post-processing (Unmap)
W	Wake the I/O-waiting process up
B	Prepare DMA-able buffer (Bounce)

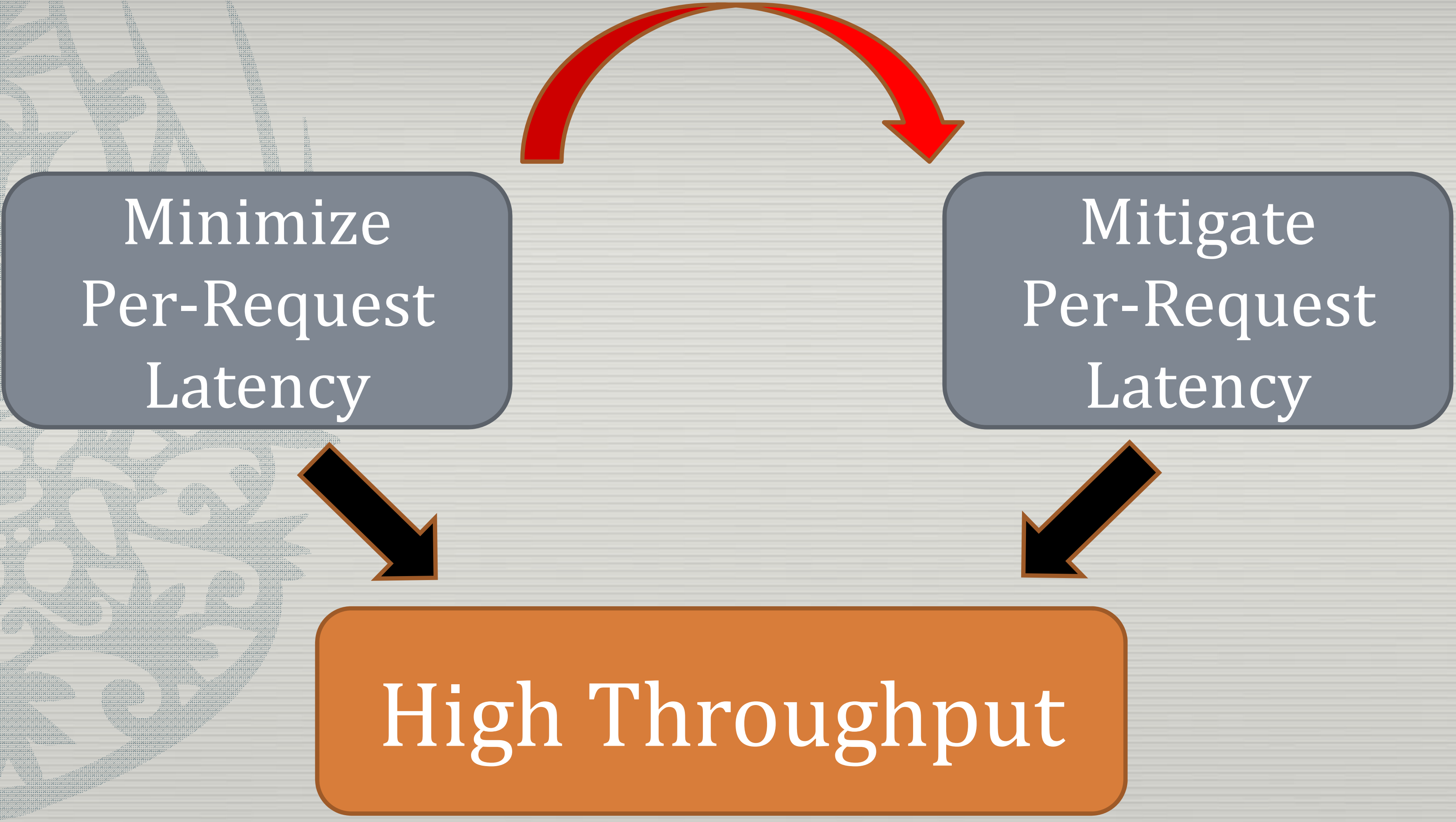
Evaluation of (Sync+Poll)



Evaluation of (Sync+Poll)

- Lesson
  ➔ Large data transfer is still important !
  - How to make a large request ?
    - **Read-ahead** under sequential read pattern
      - Still effective on (Sync+Poll)
    - **Request merge** under sequential write pattern
      - (Sync+Poll) cannot accumulate I/O requests
  - No way to make a large request under random access pattern !

Our Strategy has Changed !



Minimize Per-Request Latency

Mitigate Per-Request Latency

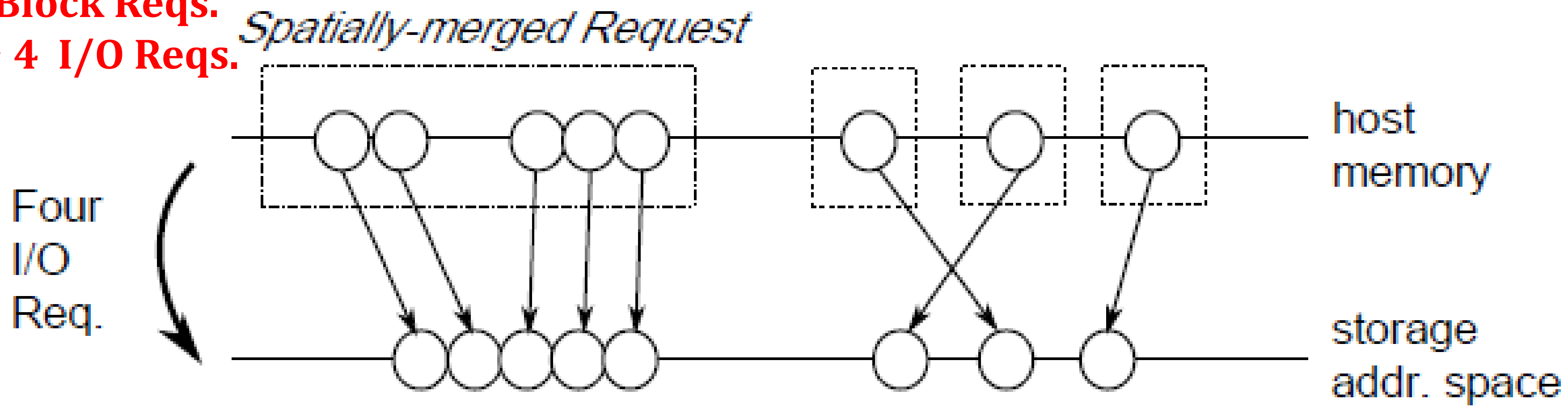
High Throughput

Solution

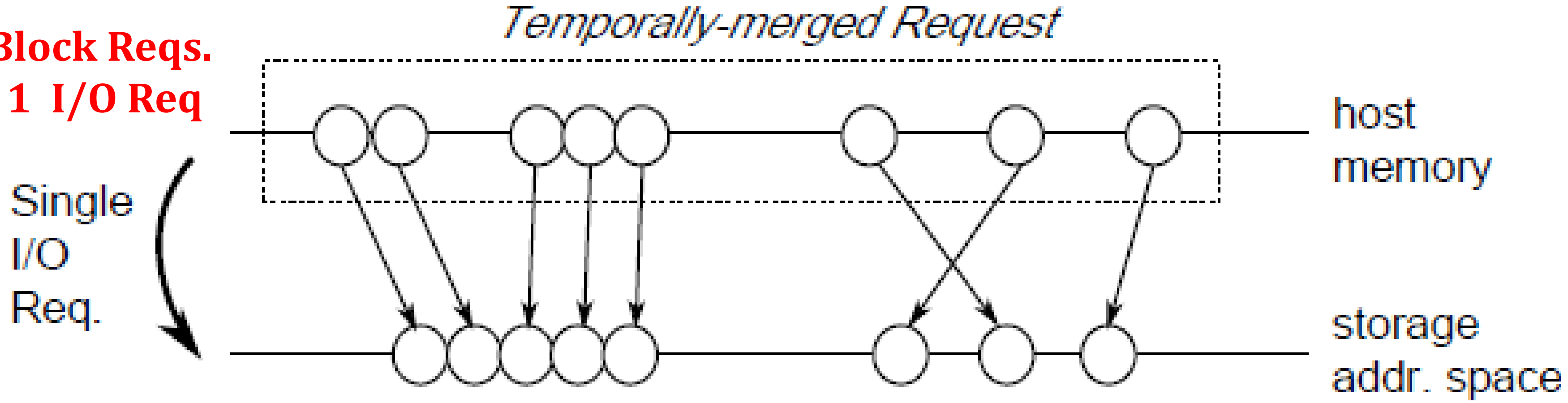
- Temporal Merge
  - Combines multiple (even non-sequential) requests within a short time window, and
  - Dispatches them by using a new I/O interface

Extended I/O Interface

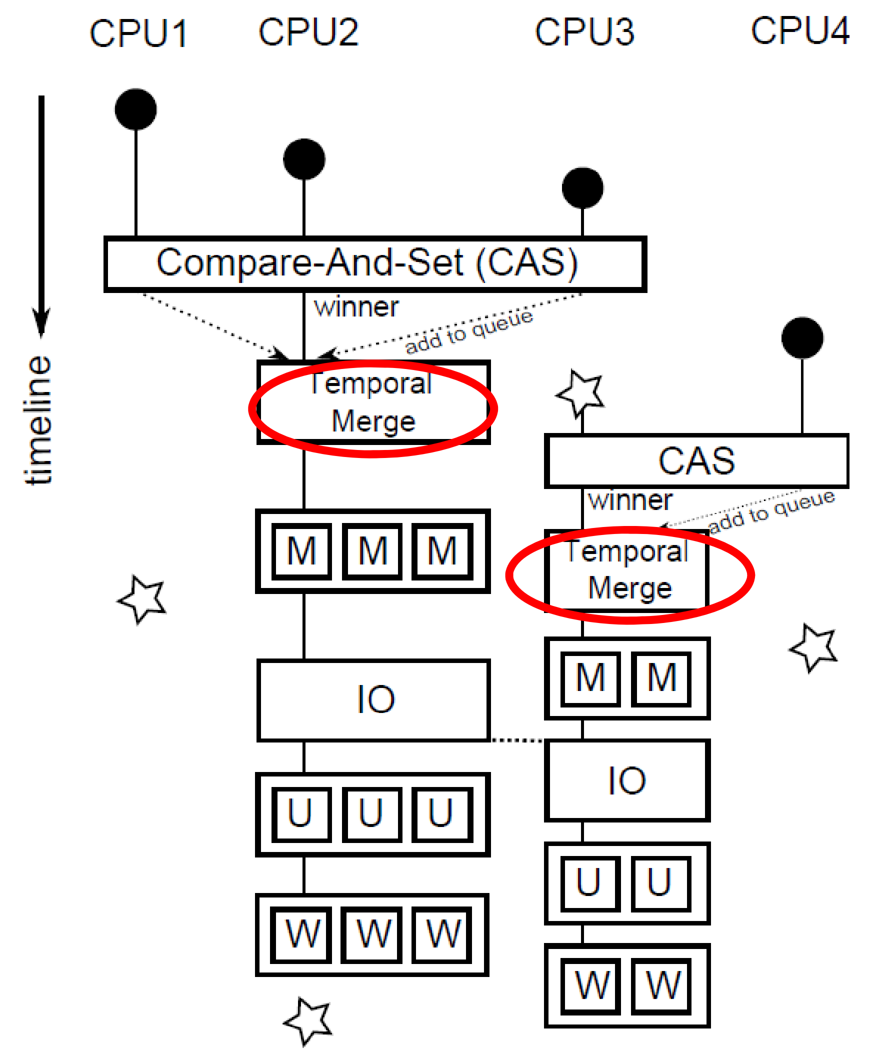
8 Block Reqs.
➔ 4 I/O Reqs.



8 Block Reqs.
➔ 1 I/O Req



Synchronous Temporal Merge



- Each thread submits a block request.
- Only one thread becomes a "winner".
- The winner combines concurrent block requests into one and dispatches it by using the new interface.
- The losing threads yield CPU and sleep until the completion of their requests.
- **Synchronous Temporal Merge**
  - No plugging/unplugging is required during merge operation.

Synchronous Temporal Merge

- Advantage
  - Balance of Synchronous I/O path and Batching
    - Low-latency (No sleep/wakeup for a winner)
    - High-throughput (Oblivious to block access pattern)

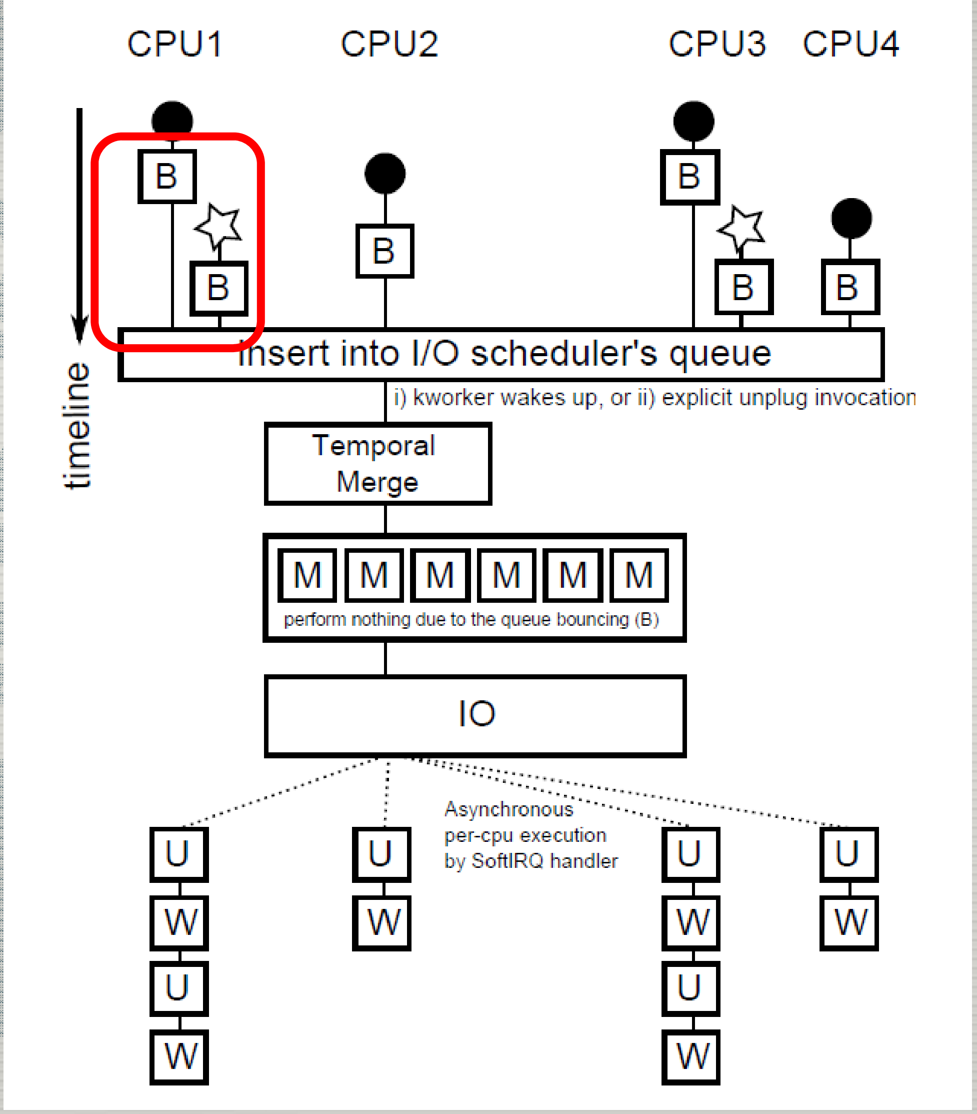
- Disadvantage
  - **Merge Count (i.e. Benefit)** is limited by **Concurrency**.
    - Concurrency: the maximum number of threads entering into I/O subsystem
    - Due to 'delayed write' semantics, the concurrency is usually lower than the number of user threads that issued write requests.

Asynchronous Temporal Merge

- How to merge I/O requests even when the number of I/O threads is very low?
  - Utilize I/O scheduler again,
  - But this time, do it with “the extended I/O interface”

- The result would depend on tradeoff bet’n
  - The advantage of large data transfer
  - The disadvantage of increased latency

Asynchronous Temporal Merge



- Each thread piles up I/O requests in a request queue.
- “kblockd” or “user process”
  1) fetches all the block requests,
  2) merges them,
  3) dispatches the merged request
- Cache-friendly request retirement by using SoftIRQ (instead of Inter-Processor-Interrupt used in MSST’10)
- Tune a few parameters
  - unplug_thresh, scheduler, …
- **Asynchronous Temporal Merge**
  - Use plugging/unplugging
  - Effective even when the concurrency is low

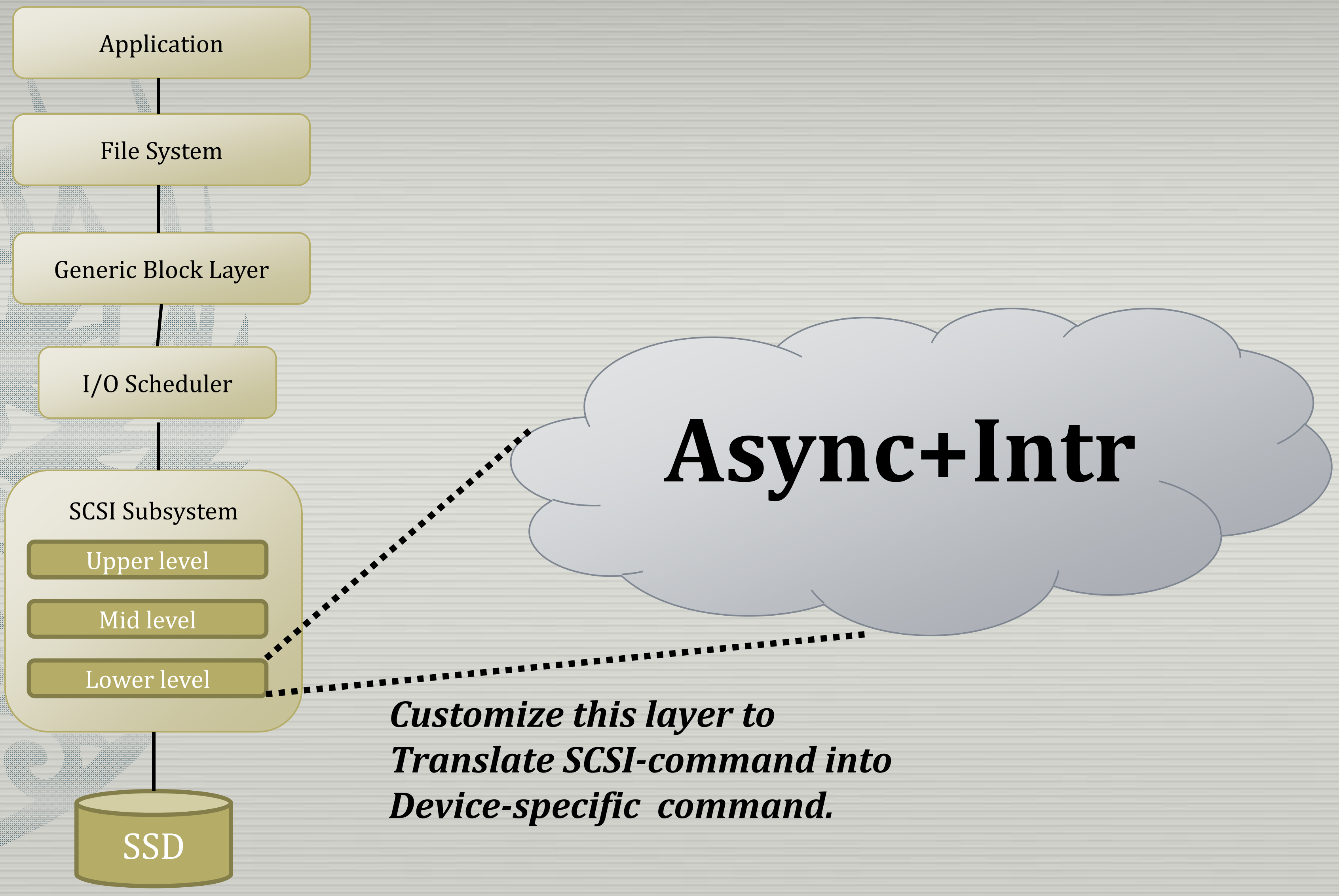
Asynchronous Temporal Merge

- Advantage
  - It could maximize the accumulation of block requests in a queue when the concurrency is low.
- Disadvantage
  - Existing I/O schedulers (in Linux) are not designed to accumulate read requests.
    - If a device is idle, a newly-arriving read request is immediately dispatched by an unplug invocation with holding a *queuelock* spinlock.

Evaluation

- Environment
  - CPU: 8 Cores (Xeon E5630@2.5GHz)
  - RAM: 2 GB (out of 16 GB) is used.
  - I/O subsystems (see next slides)
    - Async+Intr, Sync+Poll, STM+Poll, ATM+Poll
  - Benchmarks
    - Iozone, Postmark

Interrupt-based I/O Subsystem



Application

File System

Generic Block Layer

I/O Scheduler

SCSI Subsystem

Upper level

Mid level

Lower level

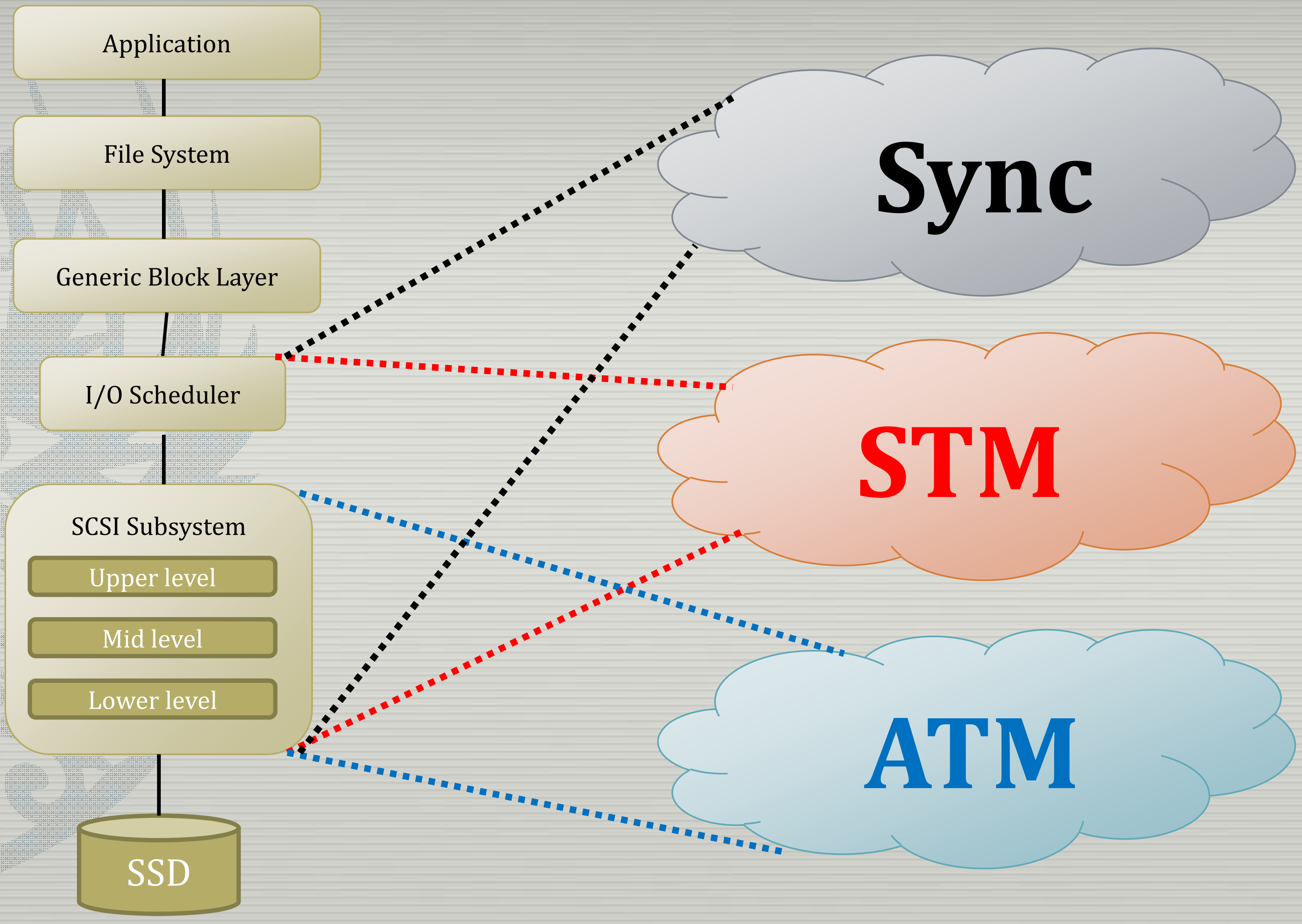
SSD

Async+Intr

Customize this layer to Translate SCSI-command into Device-specific command.

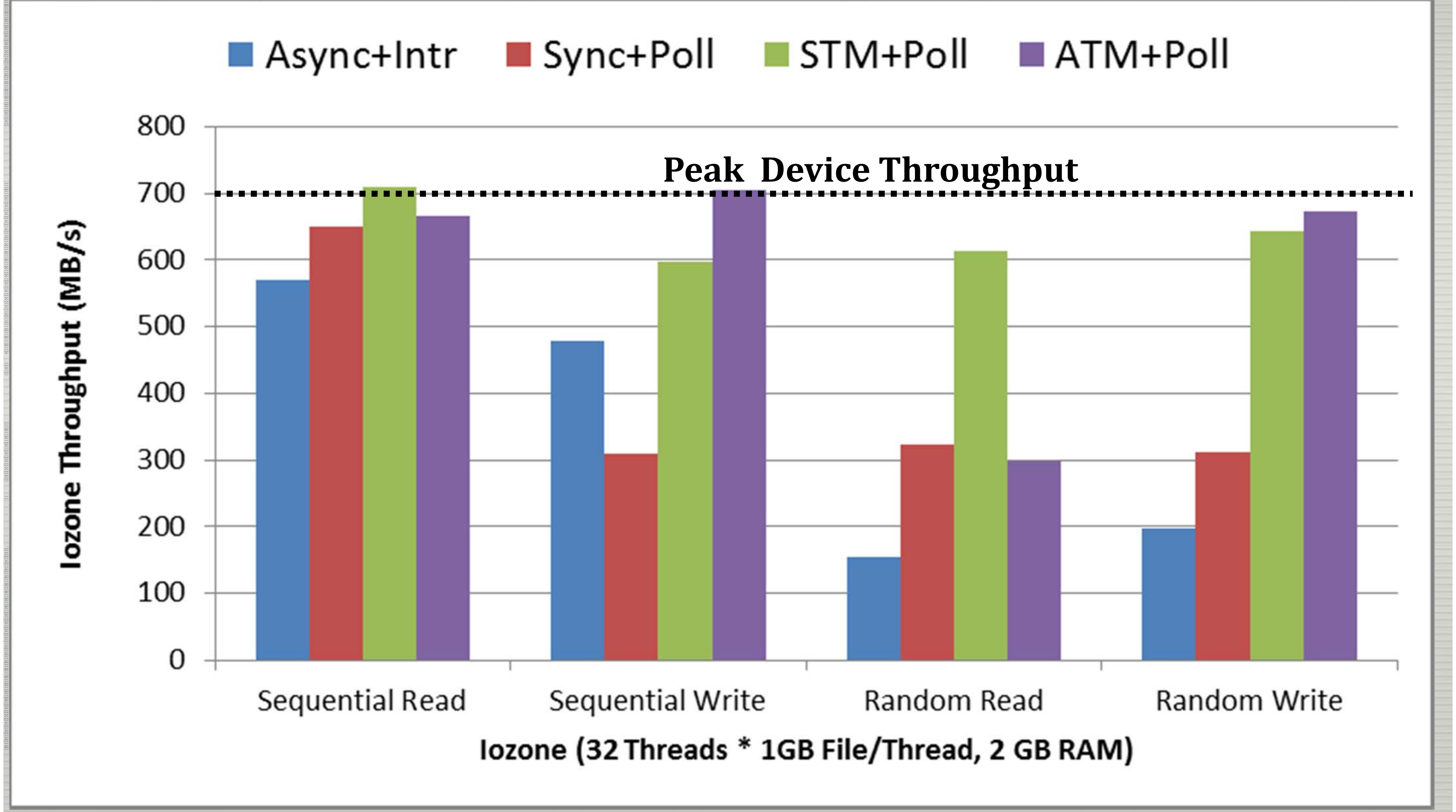
Typical Storage Stack

Poll-based I/O Subsystems



Typical Storage Stack

Evaluation
- Iozone

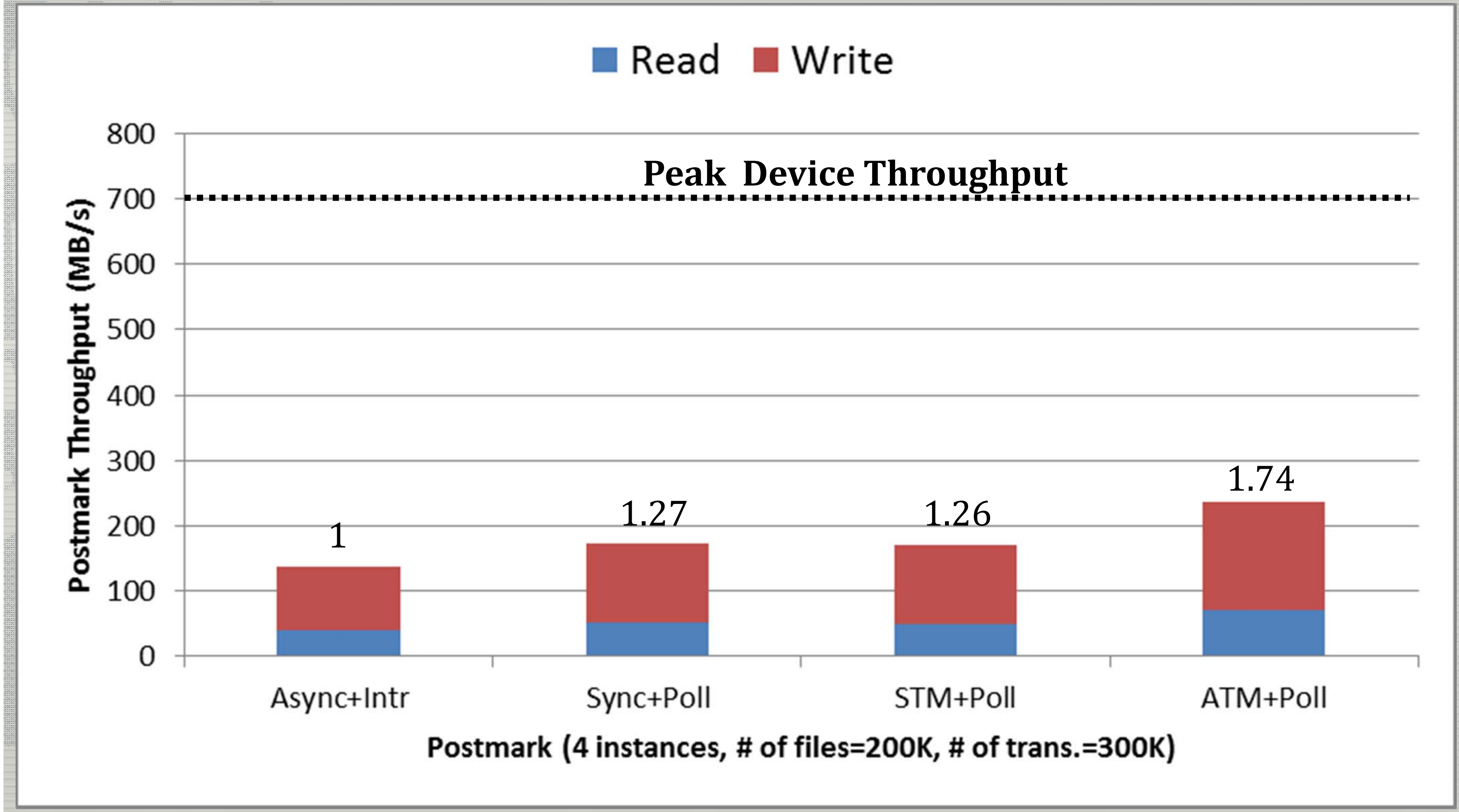


Evaluation - Iozone

	Seq.R	Seq.W	Rand.R	Rand.W
Async+Intr	82%	68%	22%	28%
Sync+Poll	93%	44%	46%	45%
STM+Poll	**100%**	**85%**	**88%**	**92%**
ATM+Poll	**95%**	**100%**	43%	**96%**

- STM achieves 85%~100% of the peak device throughput.
- ATM achieves 95%~100% of the peak device throughput except for the Random-Read access pattern.

Evaluation
- Postmark



Conclusion

- Temporal Merge
  - Enables I/O subsystem to dispatch discontiguous block requests by using an extended I/O interface
  - Helps to achieve near-peak device throughput from random access workload
- Future work
  - Standardization. (NVMHCI)
  - Reliability (atomic update)
  - Parallelism (RAID, storage network)
  - Hybrid solution with Flash + HDD