UnICom: A Universally High-Performant I/O Completion Mechanism for Modern Computer Systems

Riwei Pan, Yu Liang, Sam H. Noh, Lei Li, Nan Guan, Tei-Wei Kuo, Chun Jason Xue

香港城市大學
City University of Hong Kong

MOHAMED BIN ZAYED
UNIVERSITY OF
ARTIFICIAL INTELLIGENCE

Research Background

香港城市大學
City University of Hong Kong

Modern Computer System Evolution

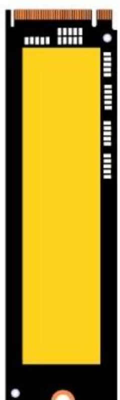
Emerging SSDs

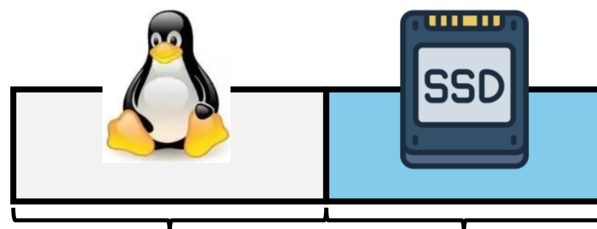
> 1500k IOPS

< 5us I/O latency

I/O Stack Overhead

up to 50% software overhead



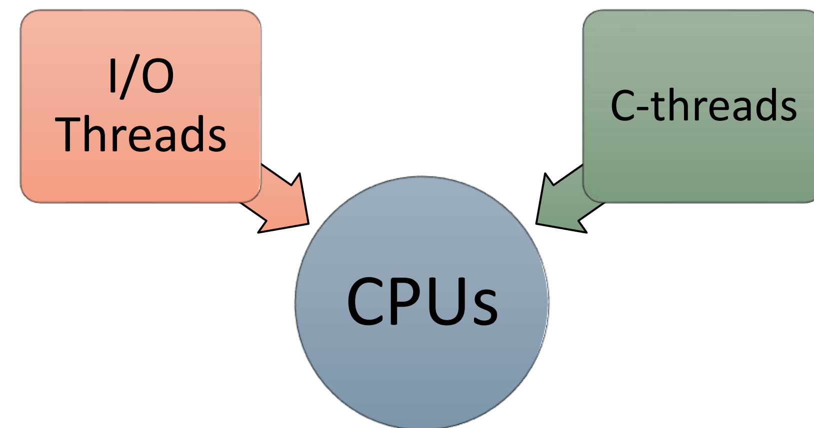


Linux I/O Stack Latency (54%)

4KB Disk I/O Latency (46%)

Universally Mixed Workloads

Coexisting I/O Threads & Compute Threads (C-threads)



I/O Completion Dilemmas of Interrupt and Polling

Dilemmas of Interrupt and Polling:

- **Interrupt**: high I/O software overhead but efficient CPU utilization
- **Polling**: low I/O software overhead but inefficient CPU utilization

Case Study:

- **Two representative works:**
  - *ext4* (interrupt)
  - *BypassD* (polling, ASPLOS'24) → enhance ext4 by a kernel-bypassing I/O path and polling.
- **Two different workloads:**
  - Pure I/O Workloads
  - Mixed Workloads
    - Co-running I/O threads and counter threads (to represent C-threads)

e.g., keep counting to simulate computation workloads

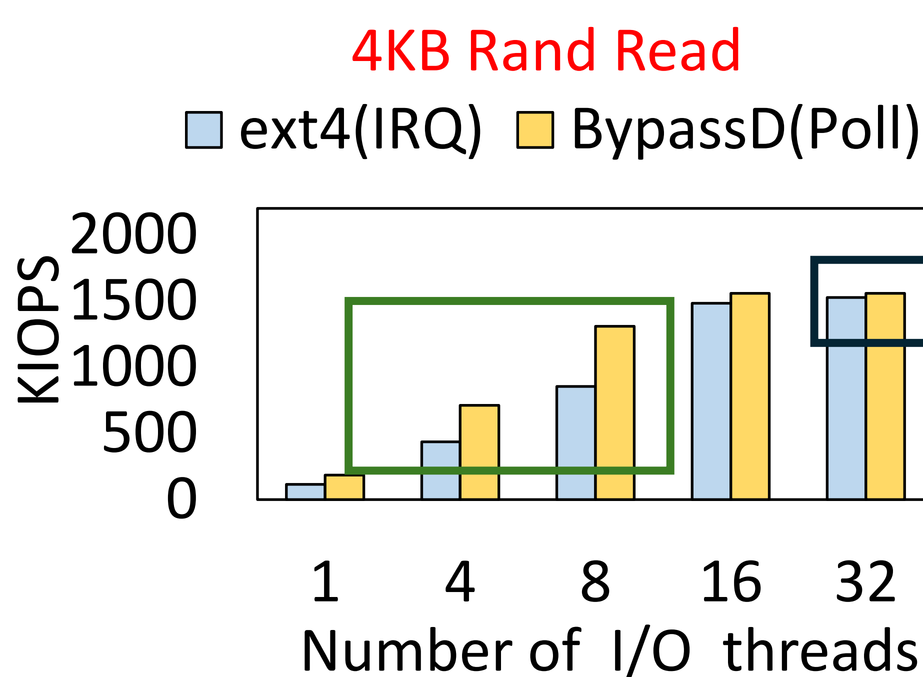
Pure I/O Workloads

CityU
香港城市大學
City University of Hong Kong

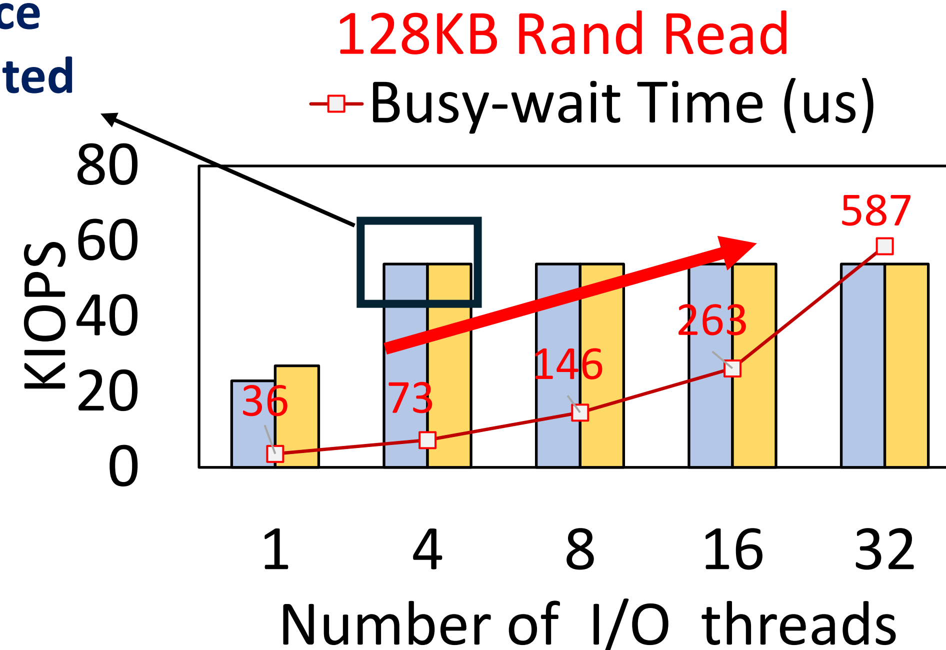
✗ **Interrupt (ext4):** *Low* IOPS performance

✓ **Polling (BypassD):** *High* IOPS performance

- For 128 KB I/Os, busy-wait time increases due to device saturation.



Device Saturated



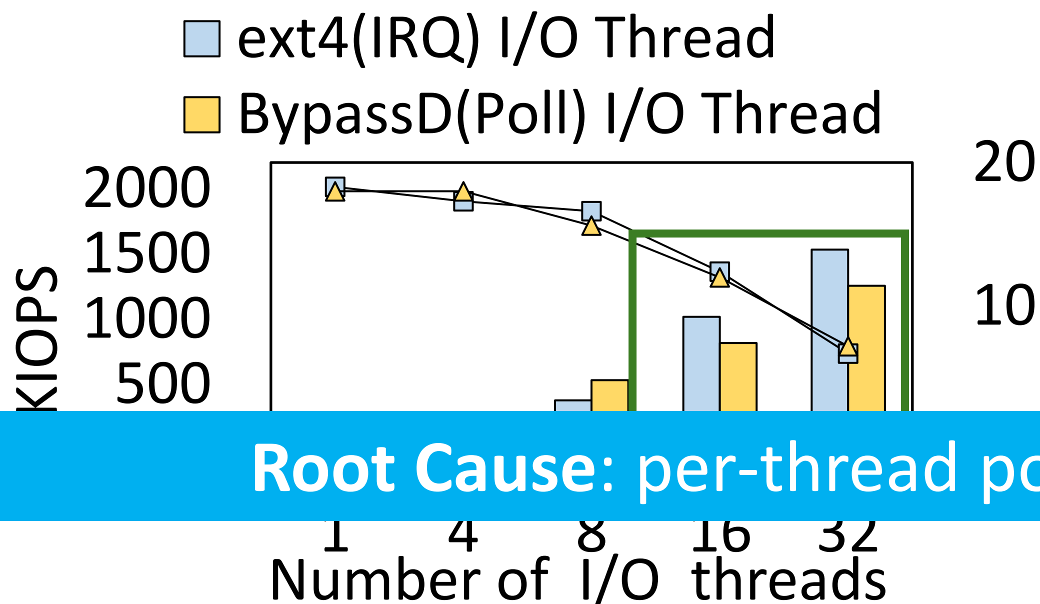
Mixed Workloads (fixed 16 C-threads)

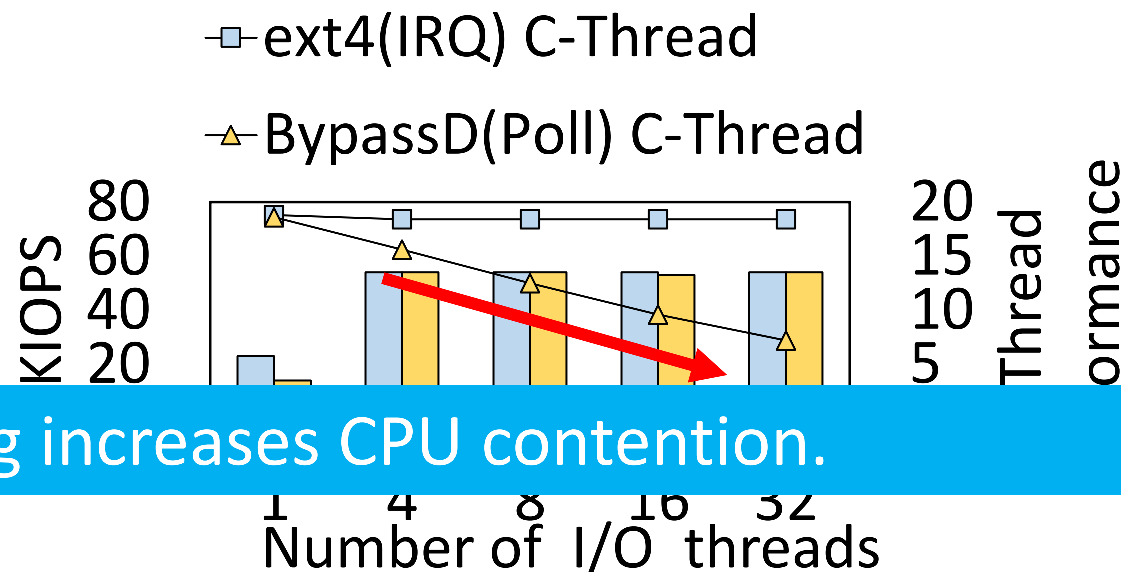
香港城市大學
City University of Hong Kong

✓ **Interrupt (ext4):** *Higher* IOPS performance and C-thread performance

✗ **Polling (BypassD):** *Both IOPS and C-thread performance is lower.*

- Cannot saturate the device due to mutual interference with 4KB I/O.
- C-thread performance continuously drops with 128 KB I/O.





Root Cause: per-thread polling increases CPU contention.

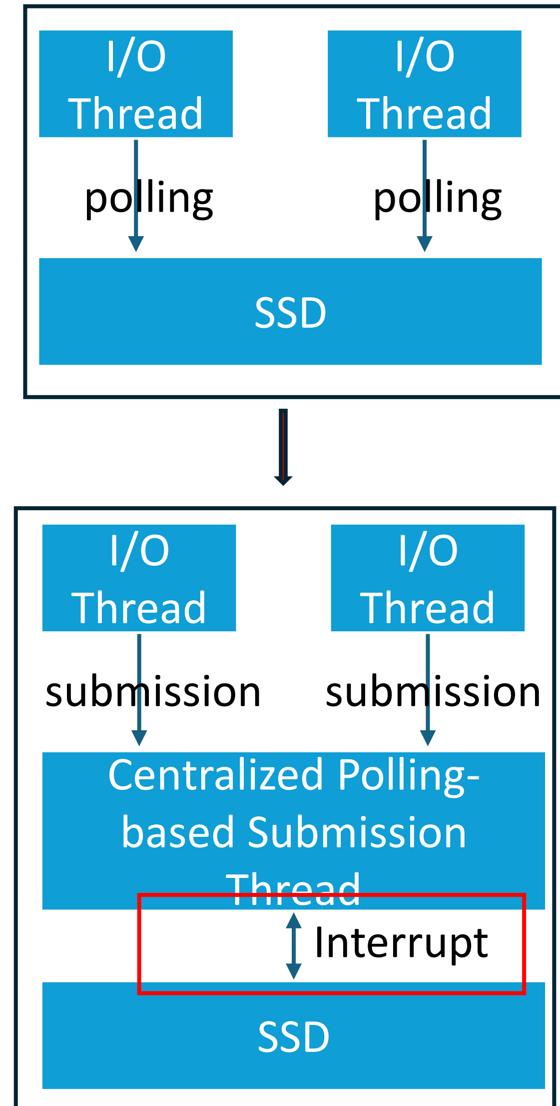
Existing Solution: Asynchronous I/O

香港城市大學
City University of Hong Kong

Observation: *Async I/O framework like **IO_uring in** SQ_POLL* mode *can merge per-thread polling into centralized polling to reduce CPU utilization.*

✓ **Consolidates per-thread polling** into a dedicated submission queue polling thread
✓ **Centralized handling of I/O completions** across different I/O threads

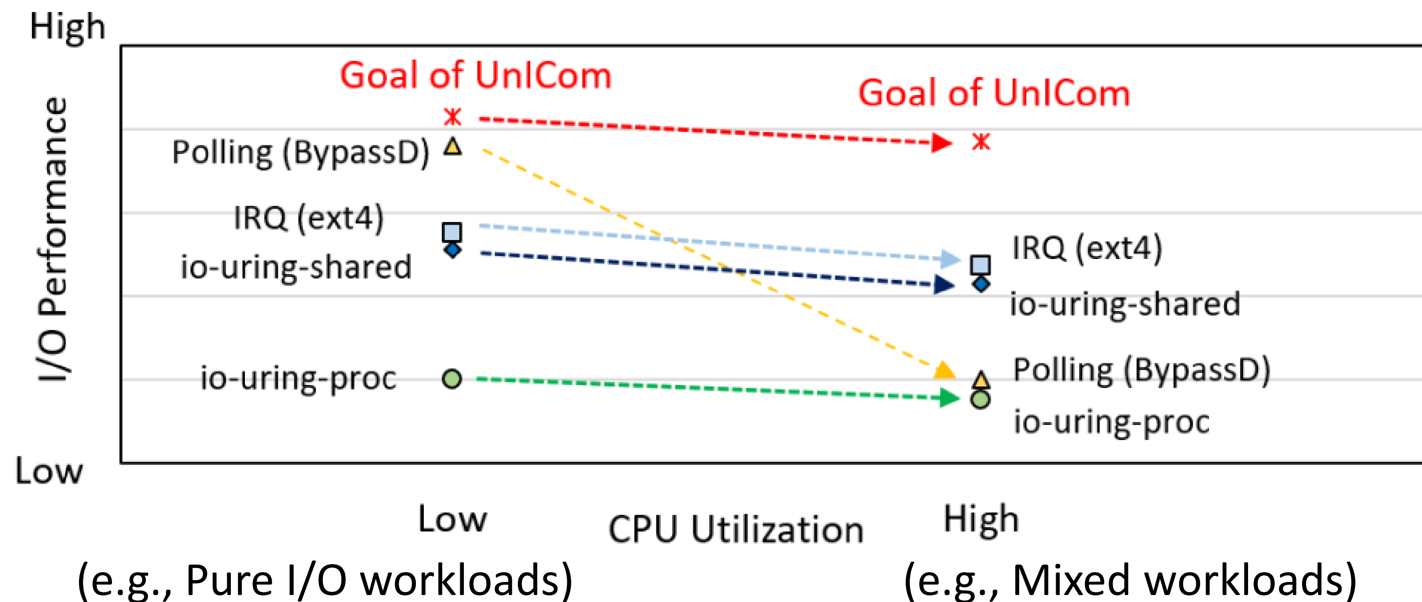
Limitations:
✗ **Incompatibility**: requires code changes for sync I/O
✗ **Performance Bound**: relies on underlying kernel completion mechanism (i.e., Interrupt)
✗ **Limited multi-process support**: per-process submission thread increases CPU contention.



Design Goal

Our goal: *designing an I/O completion mechanism that delivers universally high performance across different scenarios*:

- ✓ *Near-polling* I/O performance in pure I/O workloads (low CPU utilization)
- ✓ *Near-interrupt* I/O performance and C-thread performance in mixed workloads (high CPU utilization)
- ✓ *Supporting sync I/O* to transparently support mainstream applications



Challenges

香港城市大學
City University of Hong Kong

Near-Interrupt Performance:

- ***Challenge 1***: Heavy Sleep/Wake Overhead
- An interrupt-like sleep-wakeup mechanism is required
- Existing task deactivation and reactivation operations (e.g., enqueue and dequeue) contribute 33% overhead of the total I/O latency

Near-polling Performance and Synchronous I/O Support:

- ***Challenge 2***: Efficient polling with safe multi-process support
- A polling-based completion is required to ensure low I/O latency
- Handling I/O completions across processes with efficient CPU usage and APP transparency is difficult

Lower Software Overhead:

- ***Challenge 3***: Kernel-bypassing I/O Stack
- Direct-access to SSDs has shown its advantages on I/O performance
- Address Challenges 1 & 2 while safely and efficiently supporting kernel-bypassing I/Os is challenging

Insights

Key Design Insights:

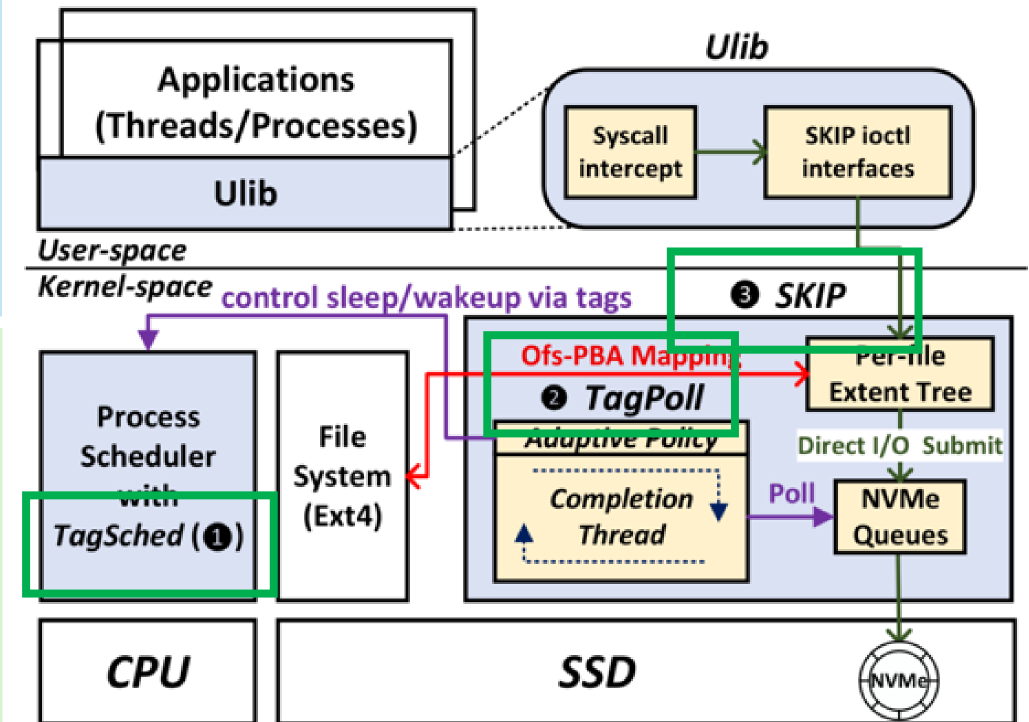
- Syscall for mode switching between user-space and kernel-space is low-cost (~1.7%).
- **Trapping into kernel to leverage kernel infrastructure to overcome Challenges 1 & 2 while bypassing most of the I/O stack to overcome Challenge 3**

4KB Disk I/O Latency		Time	Ratio
Interrupt handling	Deactivation	710 ns	8%
	Context switch	980 ns	11%
	Reactivation	1240 ns	14%
Storage device		4010 ns	46%
Syscall for mode switching		150 ns	1.7%
Others		1640 ns	19.3%
Total		8730 ns	

UnICom Solution

The key insights motivate our solution: ***Universal I/O Completion (UnICom)***

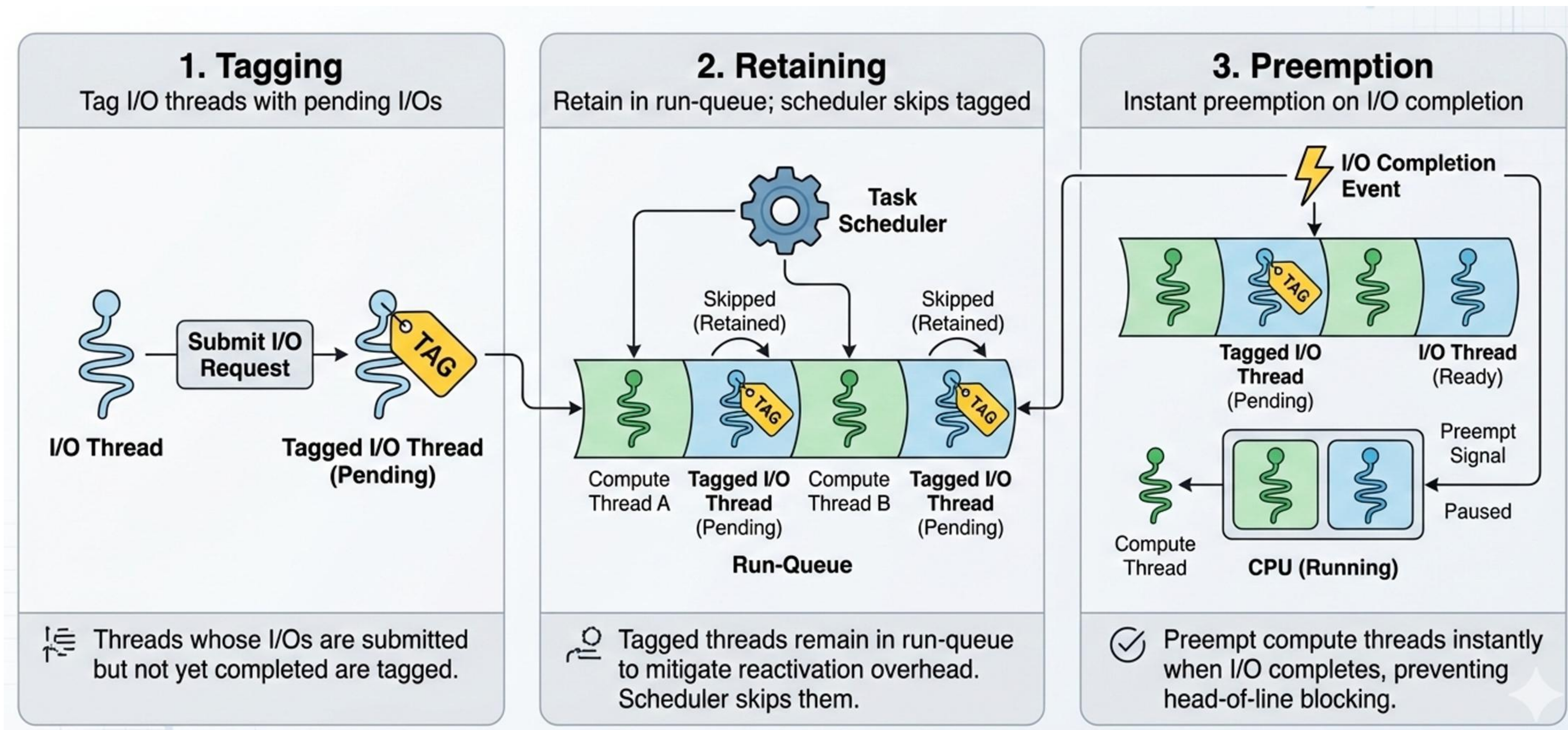
- *Challenges 1 & 2: Leveraging various kernel infrastructure:*
  - TagSched: lightweight sleeping & waking
  - TagPoll: efficient and safe polling & handling multi-process requests
- *Challenge 3: Still bypassing much of the kernel I/O stack:*
  - Shortcut Kernel I/O Path (SKIP):
    - Combining Library file system (Ulib) and Kernel module to bypass most of the kernel I/O stack
    - Translate file requests to block requests based on extent tree
    - Create an NVMe queue for TagPoll



Design Key Points: TagSched

香港城市大學
City University of Hong Kong

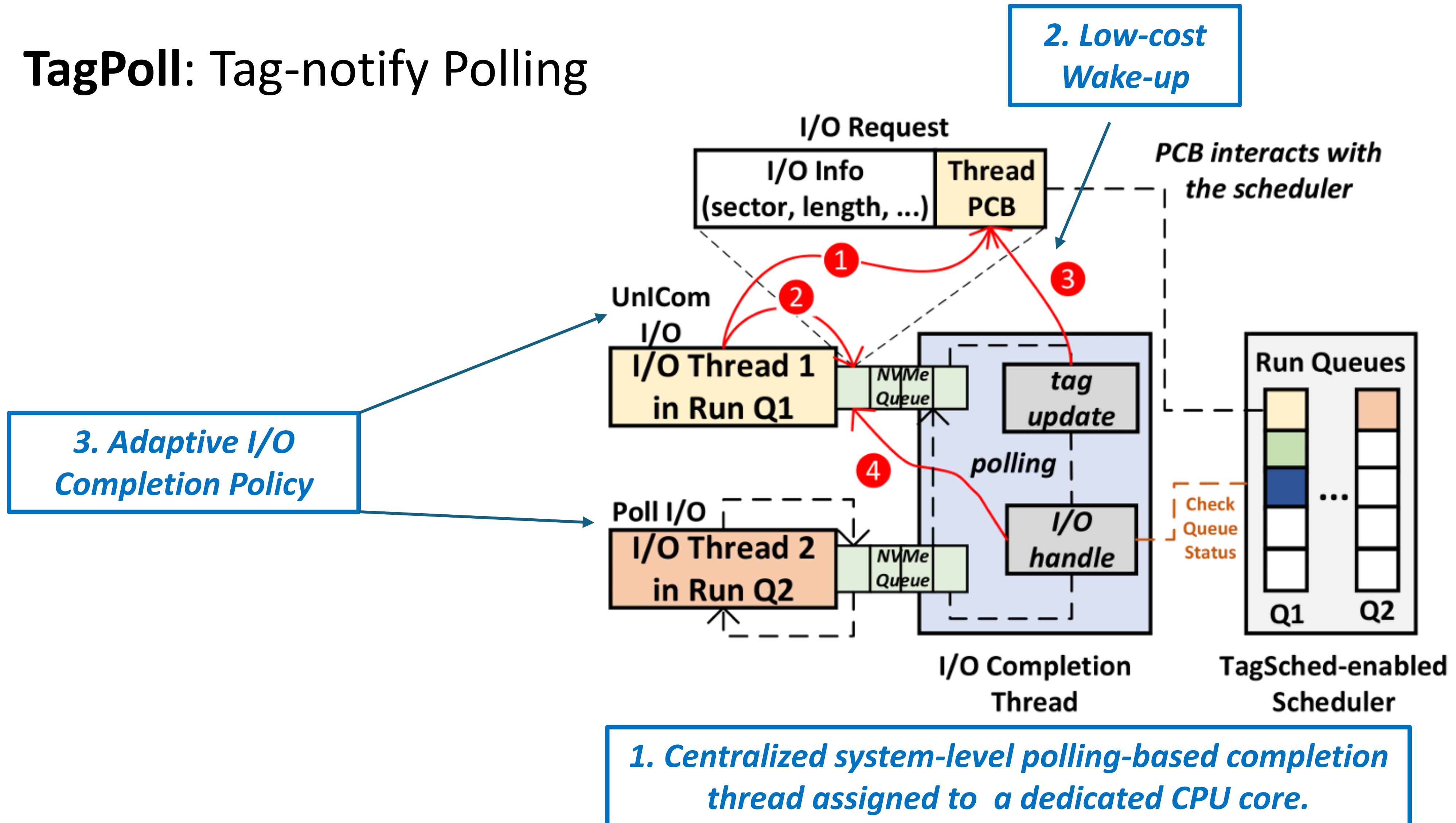
TagSched: Tag-guided in-Queue Scheduling



Design Key Points: TagPoll

香港城市大學
City University of Hong Kong

TagPoll: Tag-notify Polling



Experimental Environment

Platform: Intel Core 24-core i9-14900k, 32GB DRAM, 400GB Optane SSD P5801x

- We only use 16 E-cores of the CPU for experiments to obtain stable results.

System: Ubuntu 20.04 + Linux 6.5.1

Baselines: ext4, BypassD and io_uring in SQ_MODE

- Ext4, BypassD and io_uring use all 16 E-cores
- UnICom reserves 1 core for the dedicated I/O completion thread and the remaining 15 cores are used for applications

Evaluation: Pure I/O Workloads

Average I/O Latency

vs Interrupt (ext4):

- e.g. -42% for 4KB with one thread
- -17.4% for 128KB latency with one thread

vs Polling (BypassD):

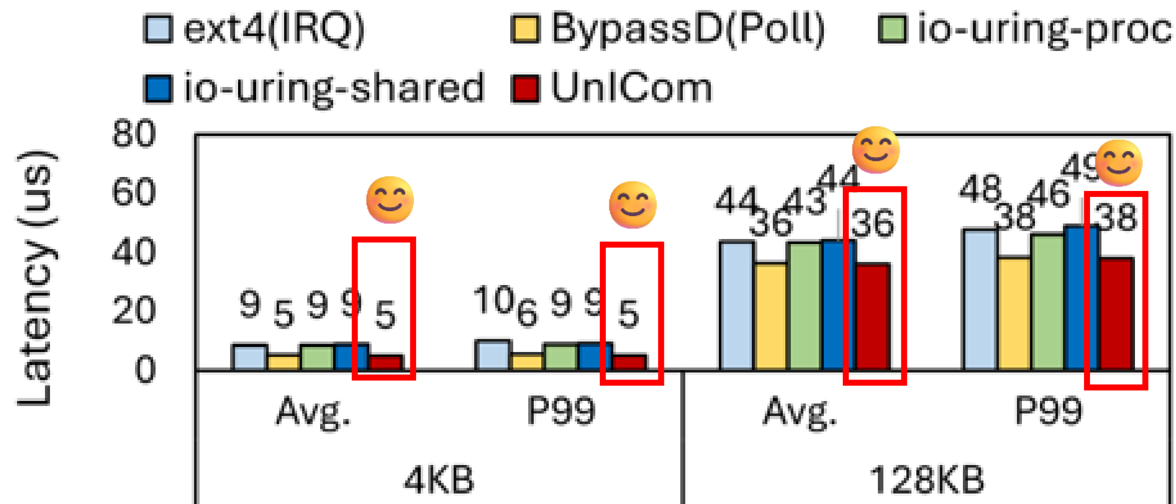
- similar performance

P99 tail latency:

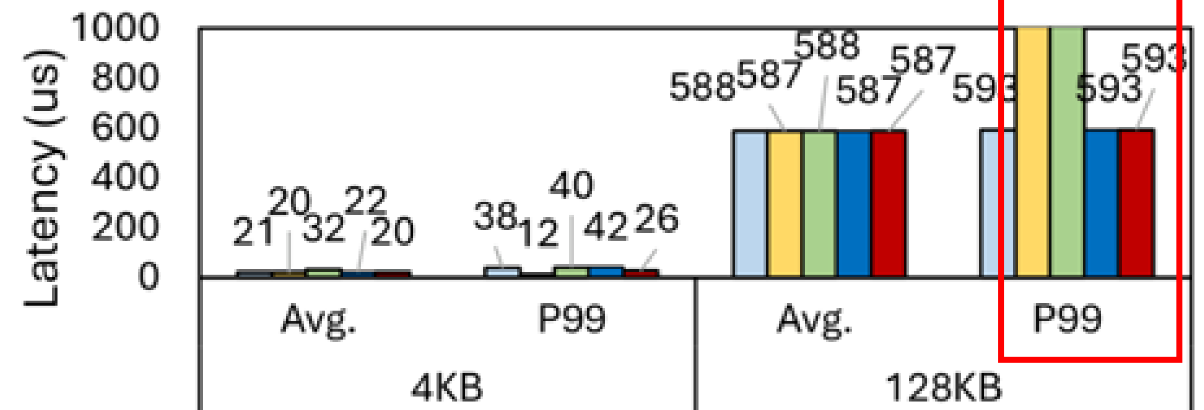
vs Interrupt (ext4):

- e.g. -31% for 4KB with 32 threads

vs Polling (BypassD): Avoid Polling's extreme long tail latency:
16175us in BypassD v.s.
~593us in UnICom for 128KB I/Os and 32 threads



(a) One I/O thread.



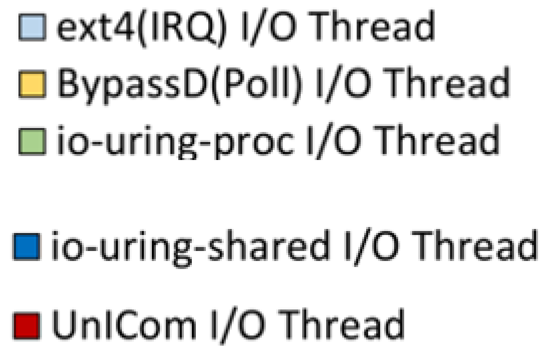
(b) 32 I/O threads.

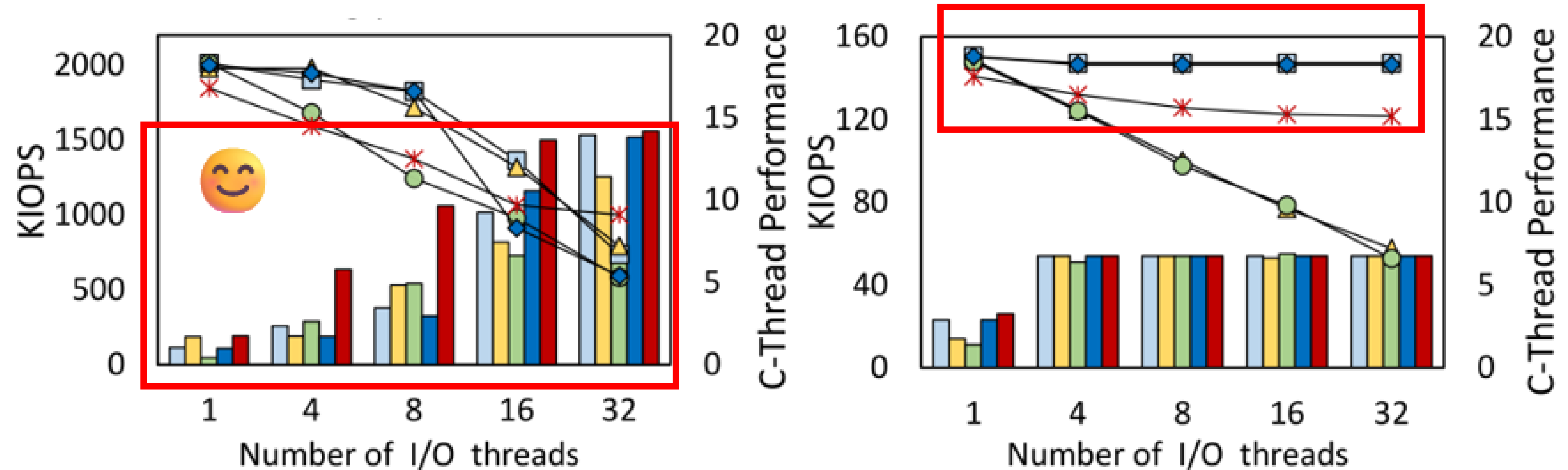
Evaluation: Mixed Workloads with C-threads

香港城市大學
City University of Hong Kong

I/O Performance with 16 fixed C-threads:

- vs Interrupt (ext4):
  - For 4KB I/Os, improve IOPS performance in low CPU utilization and similar C-thread performance in high CPU utilization (i.e., 32 I/O threads)
  - For 128KB I/Os, slightly lower C-thread performance due to the dedicated core





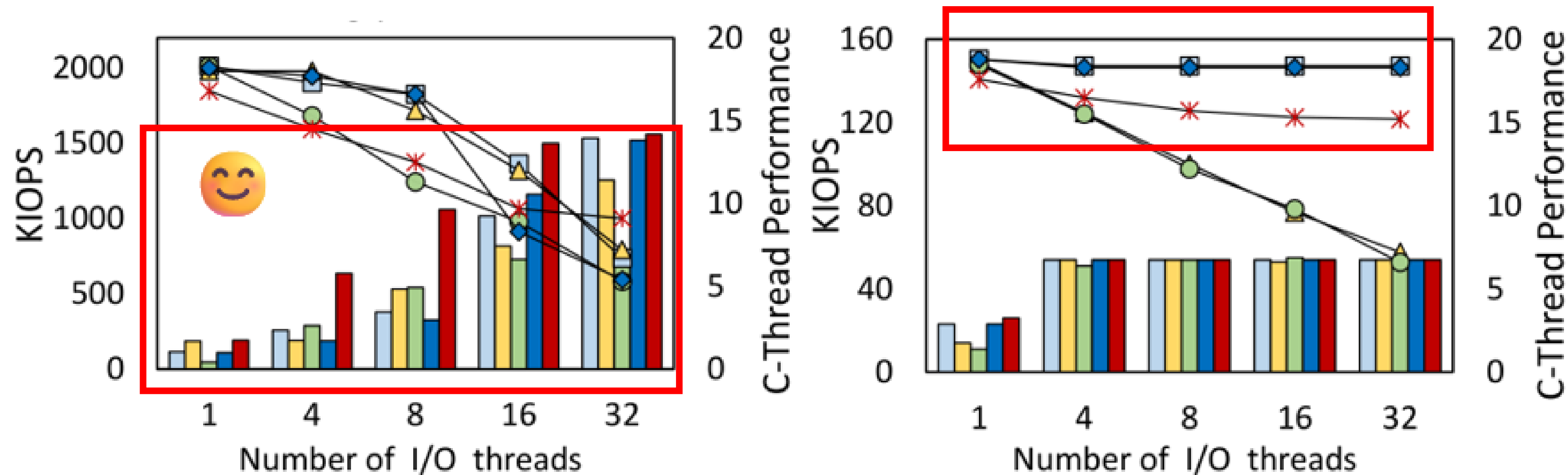
(a) 4KB rand read I/O.

(b) 128KB rand read I/O.

Evaluation: Mixed Workloads with C-threads

I/O Performance with 16 fixed C-threads:

- vs Polling (BypassD):
  - outperforms in both IOPS performance and C-thread performance



(a) 4KB rand read I/O.

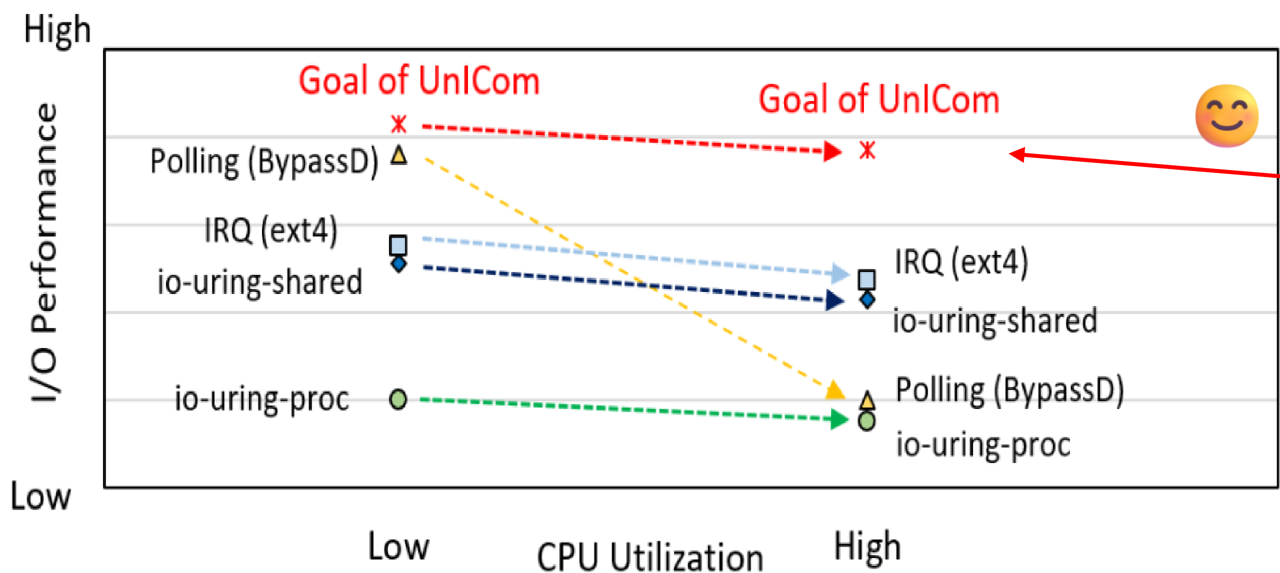
(b) 128KB rand read I/O.

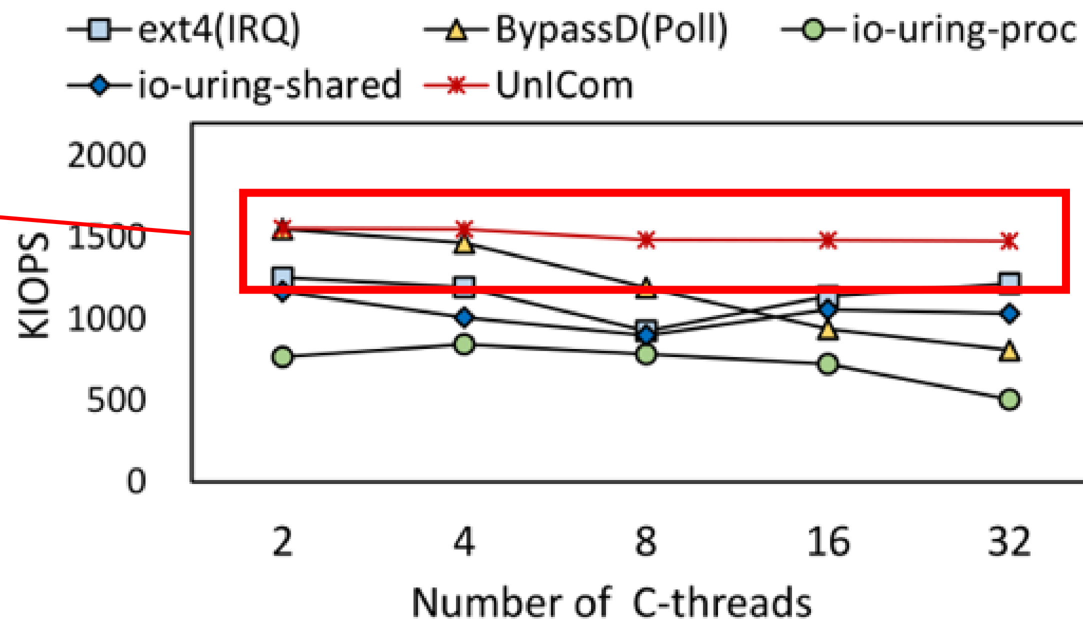
Evaluation: Goal Achievement

香港城市大學
City University of Hong Kong

Reach our goal:

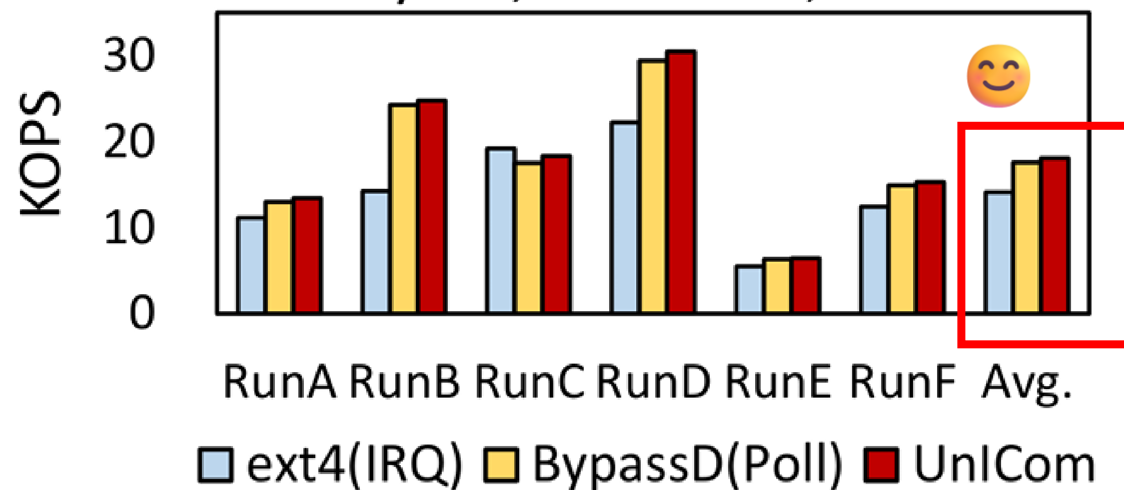
- Under different CPU utilization, UnICom shows a robust I/O performance (i.e., fixed 16 I/O threads) compared with ext4 and BypassD across different CPU utilization (i.e., the number of C-threads)



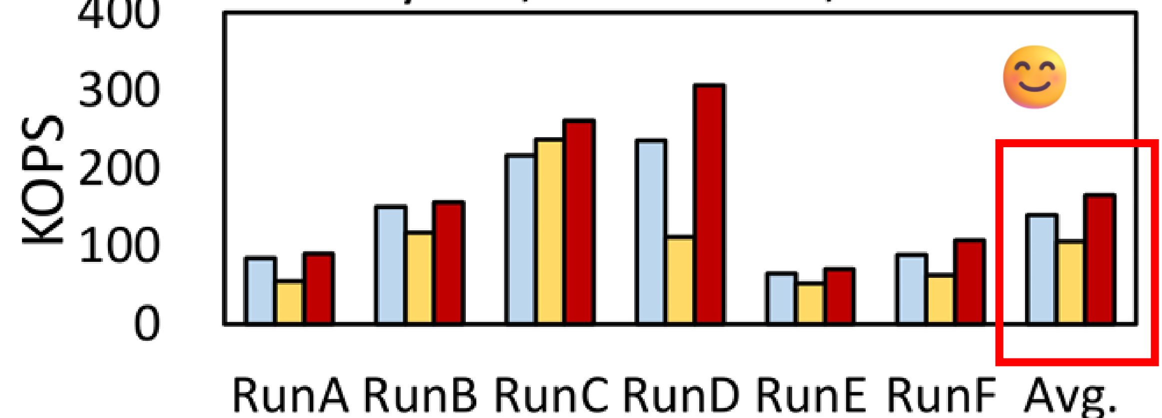


Real Application: RocksDB & YCSB

Single Thread (Average Improvement)
- vs interrupt (ext4): +28%
- vs polling (BypassD): +3%



32 Threads
- vs interrupt (ext4): +18%
- vs polling (BypassD): +56%



Conclusion

香港城市大學
City University of Hong Kong

- **Analyze the impact of existing I/O completion mechanisms across different workloads**
  - Polling will suffer from significant overhead from C-thread, and vise versa
  - Interrupt introduce significant software overhead
- **TagSched, a light-weight sleep-wakeup mechanism for I/O completion**
  - Low overhead, easy implementation
- **TagPoll, a centralized system-level polling-based I/O completion mechanism**
  - Fast I/O responsiveness, efficient CPU usage, and multi-process support
- **SKIP, a framework that efficiently and safely bypasses most of the kernel I/O stack**
- **Robust IO performance improvement across different types of workloads.**

Thanks for Listening!!

UnICom source code: https://github.com/MIoTLab/UnICom

香港城市大學
City University of Hong Kong

VIRGINIA TECH.

MOHAMED BIN ZAYED
UNIVERSITY OF
ARTIFICIAL INTELLIGENCE