USENIX ATC 23

Light-Dedup:

A Light-weight Inline Deduplication Framework for Non-Volatile Memory File Systems

Jiansheng Qiu*, **Yanqi Pan***, Wen Xia, Xiaojia Huang, Wenjun Wu,
Xiangyu Zou, Shiyi Li, Yu Hua
(*: co-first authors)

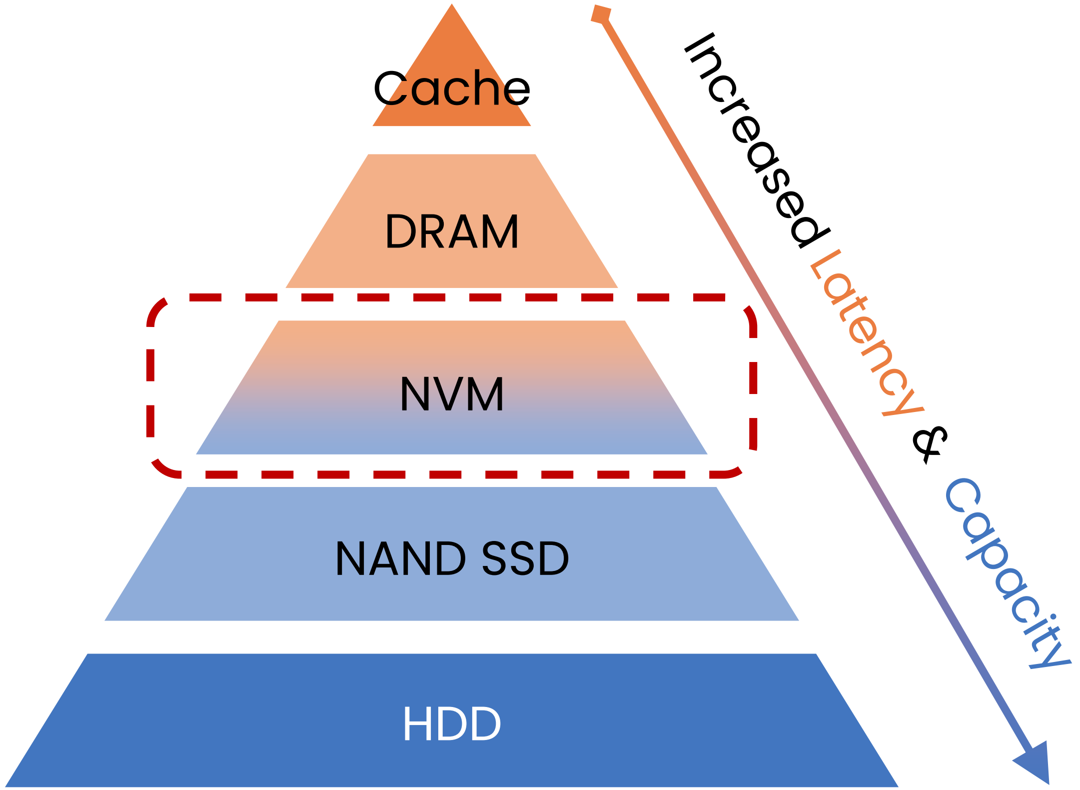
哈爾濱工業大學(深圳)
HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

華中科技大學
HUAZHONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Why NVM Dedup is Important?

- **NVM promises to be the next-generation storage media**
  - ✓ Memory Interface
  - ✓ Much Faster than SSDs/HDDs
  - ✓ Persistence, Denser (but Slower) than DRAM
- **However, NVM is expensive**
  - Intel Optane DC Persistent Memory Module* ≈ 8.8 $/GiB
  - **31×** - Intel SSD 760p ≈ 0.28 $/GiB
  - 303× - Seagate BarraCuda ≈ 0.029 $/GiB

*The only commercially available NVM

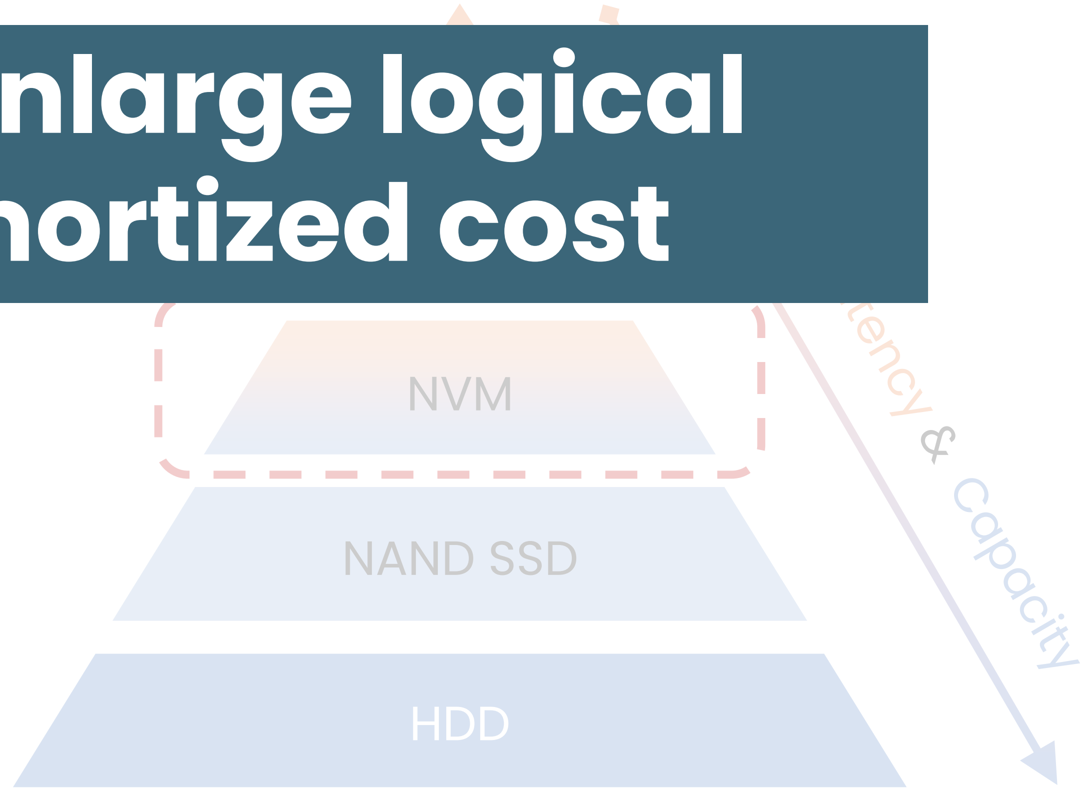


Why NVM Dedup is Important?

- **NVM promises to be the next-generation storage media**
  - ✓ Memory Interface

Deduplication can enlarge logical space & reduce amortized cost

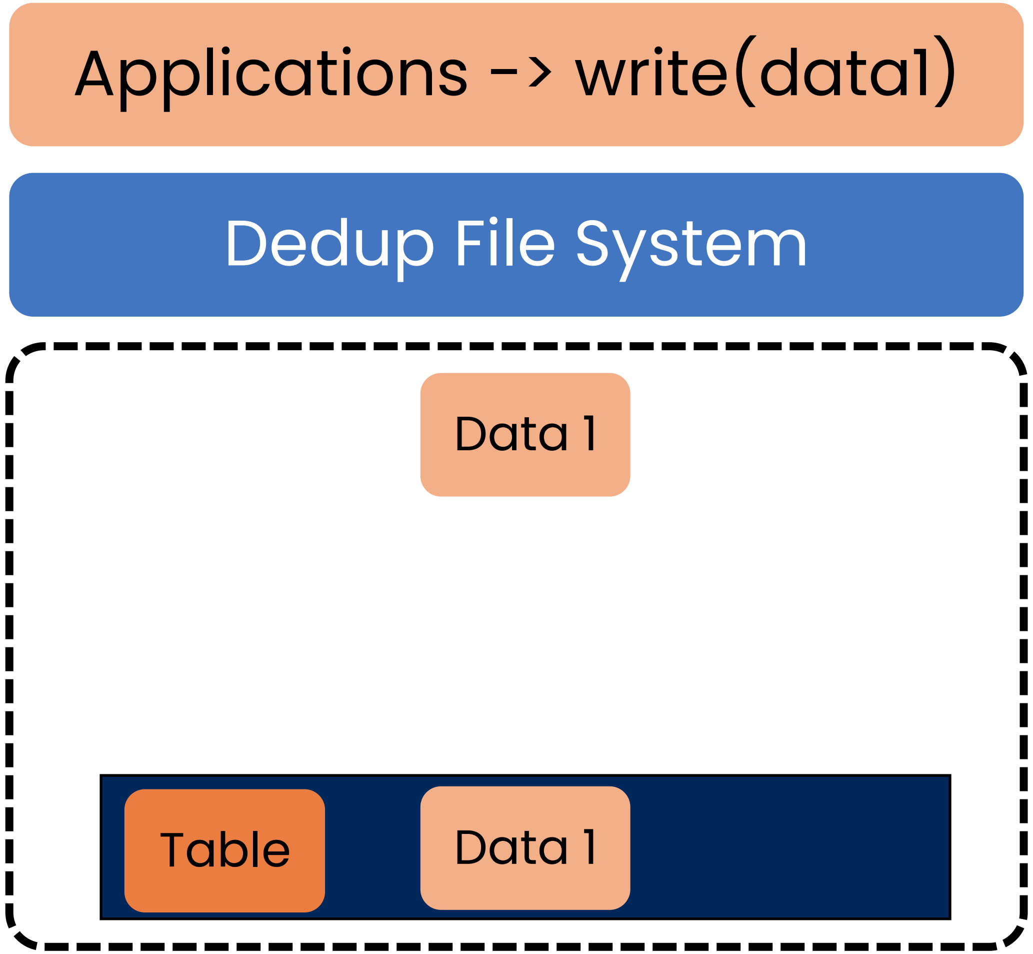
- **However, NVM is expensive**
  - Intel Optane DC Persistent Memory Module* ≈ 8.8$/GiB
  - 31× - Intel SSD 760p ≈ 0.28$/GiB
  - 303× - Seagate BarraCuda ≈ 0.029$/GiB



*The only commercially available NVM

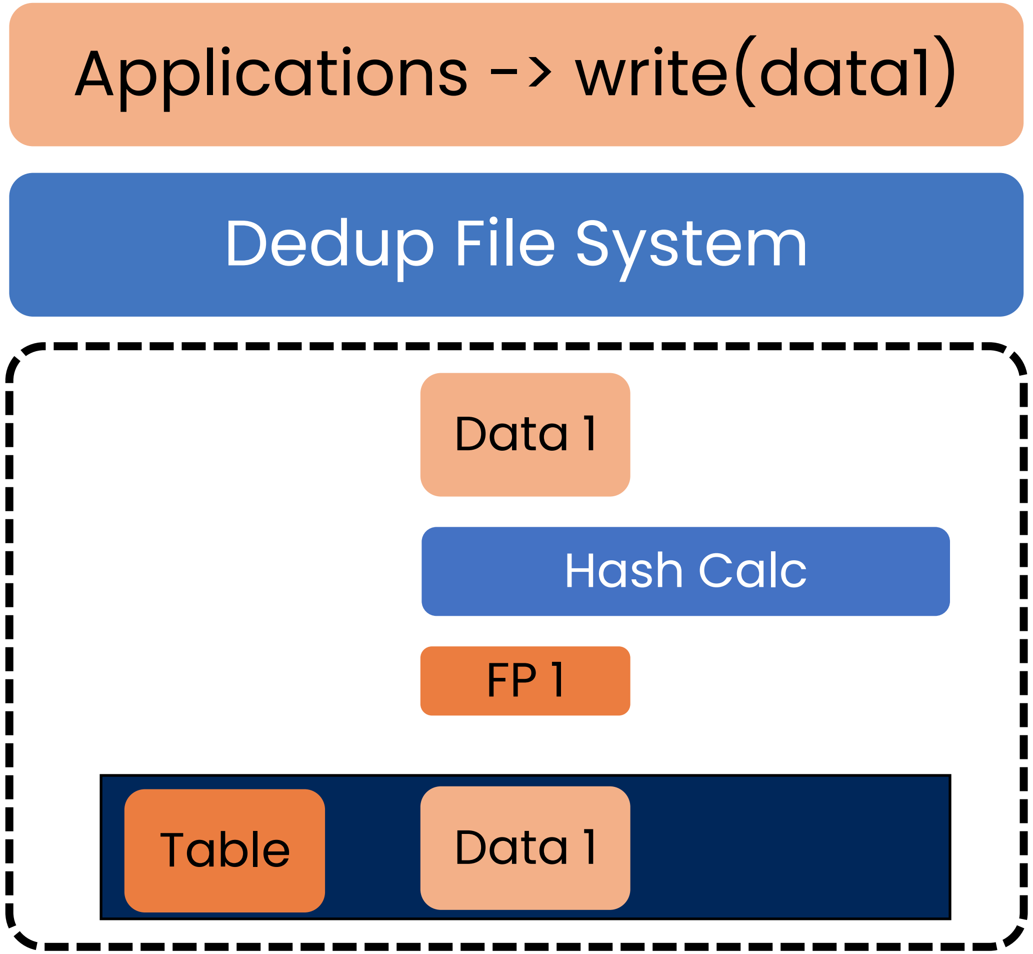
Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data



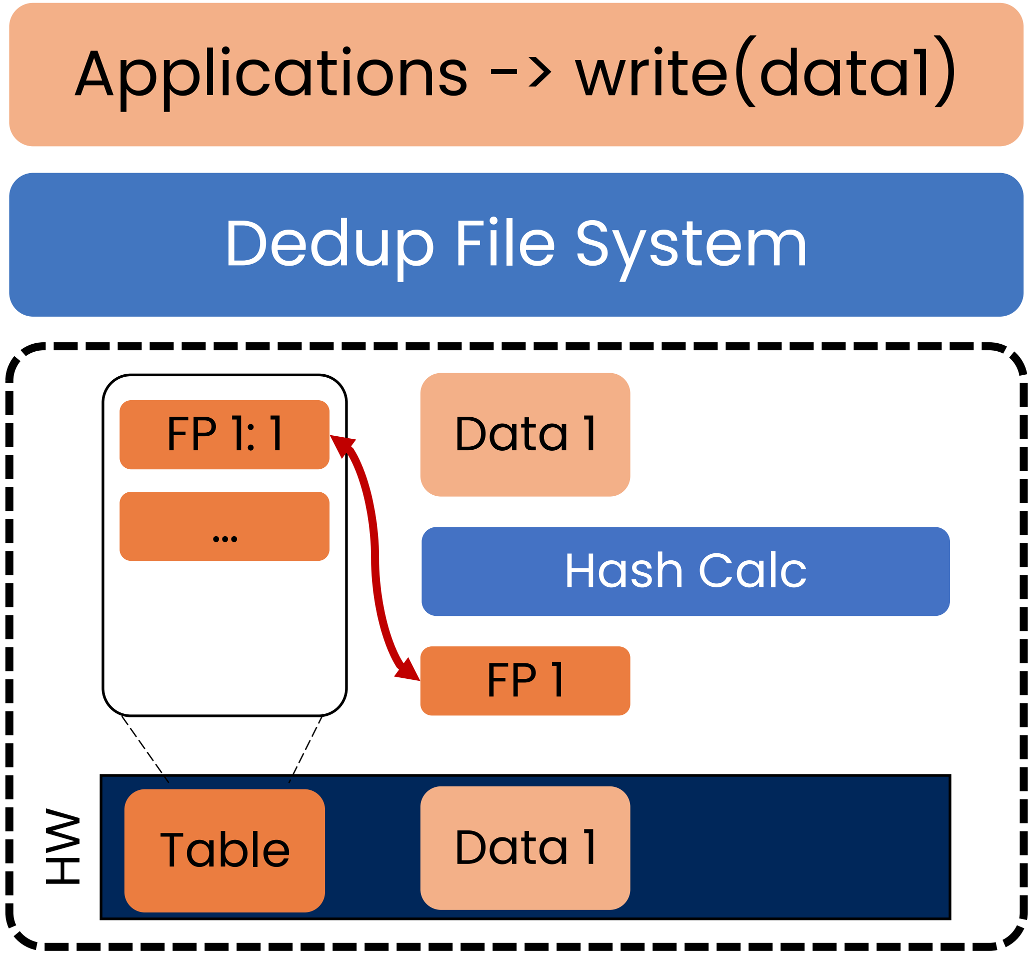
Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints



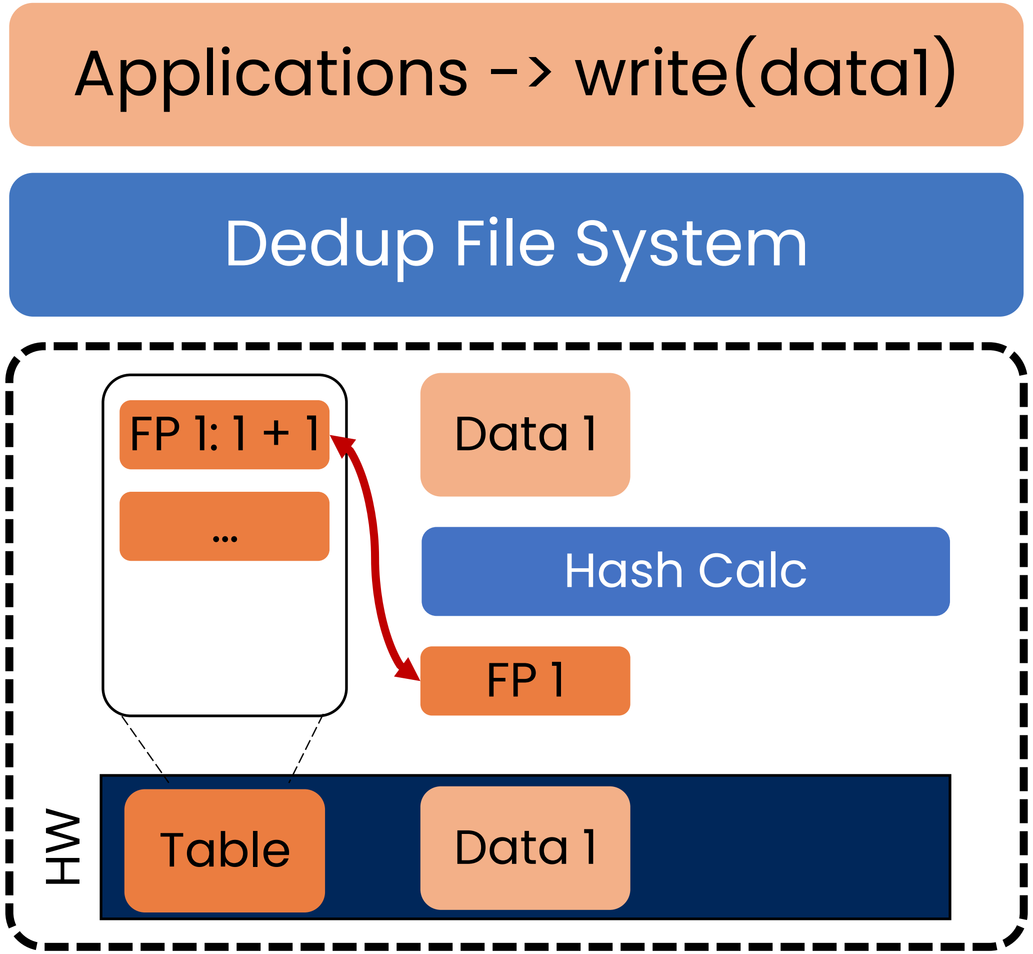
Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints
  3. Using **fingerprints** to determine if the data block is redundant



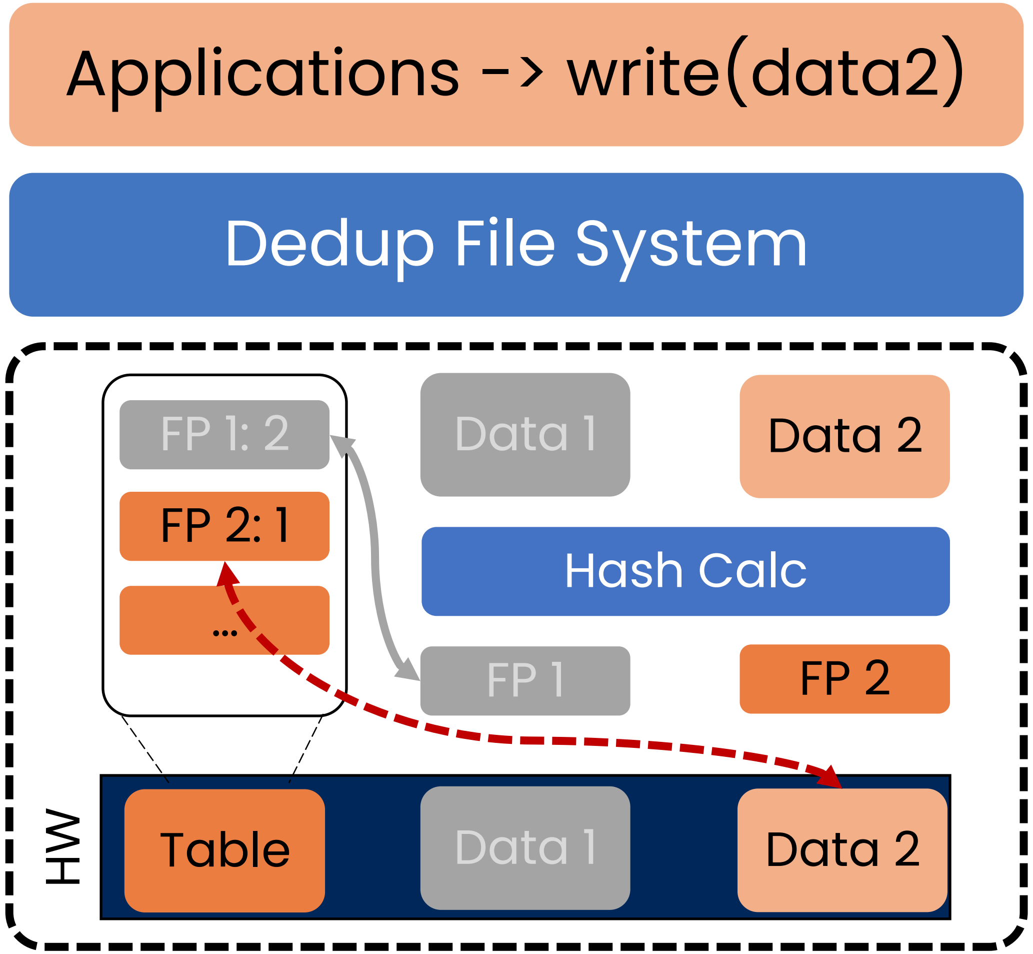
Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints
  3. Using **fingerprints** to determine if the data block is redundant
  4. If so, just modify reference count



Workflow of Deduplication File System

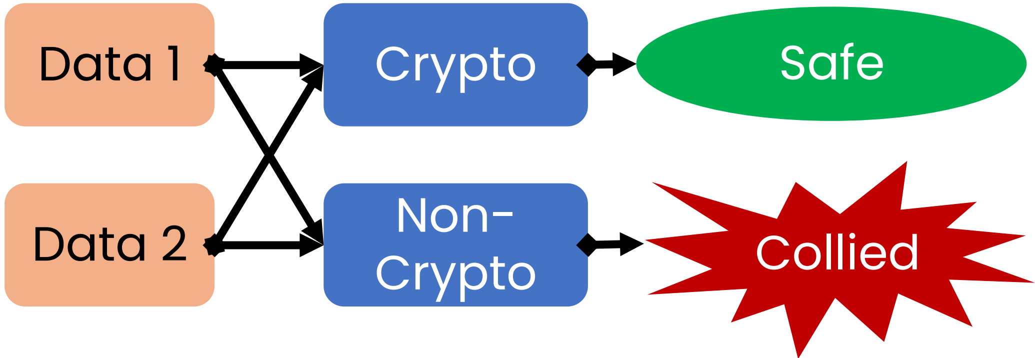
- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints
  3. Using **fingerprints** to determine if the data block is redundant
  4. If so, just modify reference count
  5. Otherwise, write the data with reference count equals 1



Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints
  3. Using **fingerprints** to determine if the data block is redundant
  4. If so, just modify reference count
  5. Otherwise, write the data with reference count equals 1

- **Non-cryptographic Hash**
  - Not safe. Need content-comparison (e.g., xxHash).
  - Light-weight calculation
- **Cryptographic Hash**
  - Safe. No additional I/O (e.g., SHA256).
  - Slow calculation



Workflow of Deduplication File System

- **Inline Deduplication**
  1. Apps write() data
  2. Calc hash as fingerprints
  3. Using **fingerprints** to determine if the data block is redundant
  4. If so, just modify reference count
  5. Otherwise, write the data with reference count equals 1

- **Offline Deduplication**

  Similar to inline deduplication, but in the background, i.e., **data must be written first**

Deduplication on NVM File Systems

- **NVM changes the game of deduplication**
  - × **Offline deduplication** can neither improve I/O performance nor lifetime of NVM
  - ✓**Using Inline deduplication** to timely eliminate redundancy and improve NVM's lifetime

  - × **Cryptographic-hash-based fingerprint** cannot well apply to fast NVM since NVM alters software and I/O bottlenecks
  - ✓**Using non-cryptographic-hash-based fingerprint** with byte-by-byte content-comparison to enable quick calculation

Deduplication on NVM File Systems

- **NVM changes the game of deduplication**
  - × **Offline deduplication** can neither improves I/O performance nor lifetime of NVM

However, can using non-crypto hash alone for NVM Dedup fully exploit NVM?

  - × **Cryptographic-hash-based fingerprint** cannot well apply to fast NVM since NVM alters software and I/O bottlenecks
  - ✓ **Using non-cryptographic-hash-based fingerprint** with byte-by-byte content-comparison to enable quick calculation

Exploiting NVM I/O Characteristics

- ★ Asymmetry in Read/Write Bandwidth (Yang@FAST'21, etc.)
- ★ I/O with Buffers (Xiang@Eurosys'22, etc.)
- ★ Long Media Read Latency (Xiang@Eurosys'22, etc.)
- ★ Coarse Access Granularity (Hyokeun@TOC'2019, etc.)
- ★ Memory Interface

Exploiting NVM I/O Characteristics

- ★ Asymmetry in Read/Write Bandwidth (Yang@FAST'21, etc.)
  - ✓ Using non-crypto hash with content-comparison
- ★ I/O with Buffers (Xiang@Eurosys'22, etc.)
- ★ Long Media Read Latency (Xiang@Eurosys'22, etc.)
- ★ Coarse Access Granularity (Hyokeun@TOC'2019, etc.)
- ★ Memory Interface

Exploiting NVM I/O Characteristics

- ★ Asymmetry in Read/Write Bandwidth (Yang@FAST'21, etc.)
- ★ I/O with Buffers (Xiang@Eurosys'22, etc.)
- ★ Long Media Read Latency (Xiang@Eurosys'22, etc.)
- ★ Coarse Access Granularity (Hyokeun@TOC'2019, etc.)
- ★ Memory Interface

× Fail to exploit or to be considered

Exploiting NVM I/O Characteristics

- ★ Asymmetry in Read/Write Bandwidth (Yang@FAST'21, etc.)

How can these features affect NVM Dedup FS? And how to exploit them?

- ★ Coarse Access Granularity (Hyokeun@TOC'2019, etc.)
- ★ Memory Interface

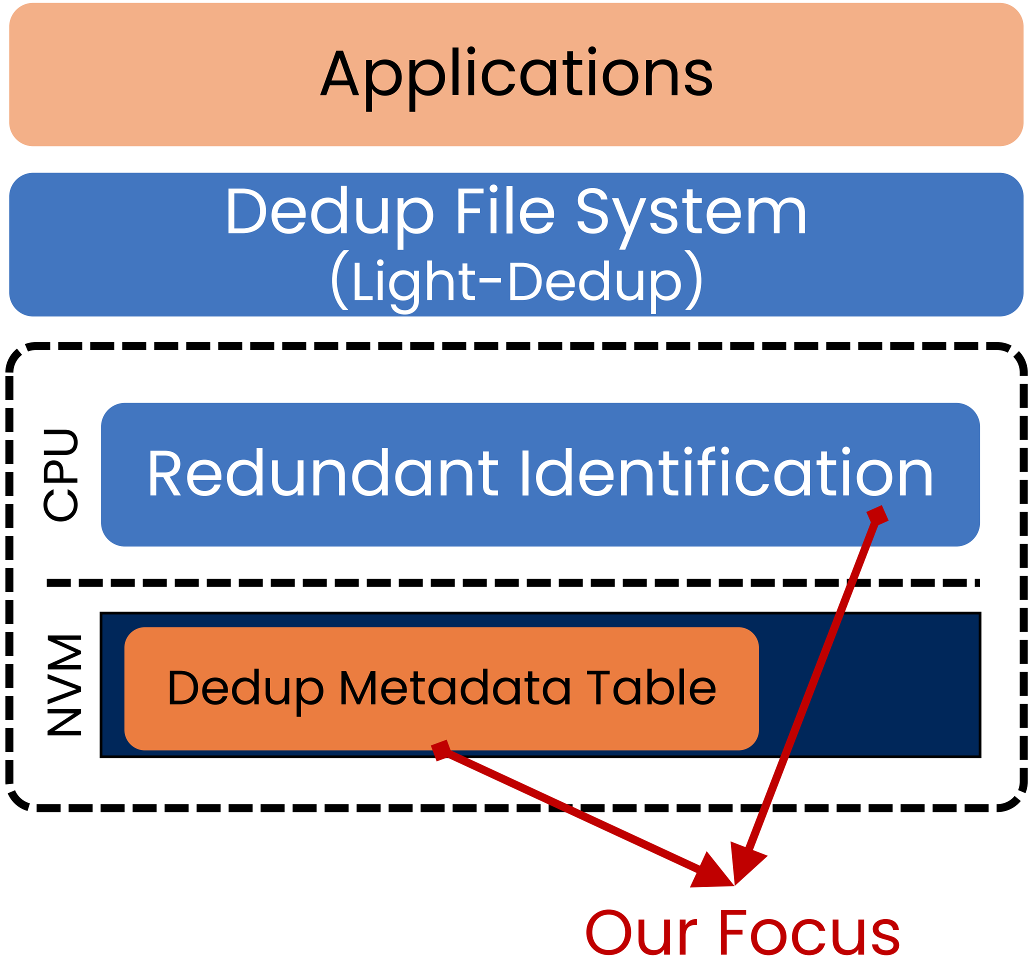
× Fail to exploit or to be considered

Goal of This Work

✓How is Dedup affected by NVM I/O features?

✓Maximize Dedup perf by fully considering NVM I/O features

✓Minimize negative impacts of Dedup for NVM file systems



Issue #1. Redundancy Identification Fails to consider I/O Buffers and Read Latency

- **Write latency can be hidden by calculation**
  - NOVA. A state-of-the-art NVM file system
  - *Naïve**. A non-crypto-hash-based Dedup file system
  - Experiment. Write two identical 4GiB files

During the first write

Much less write time!

System	Calc Latency (ns)		I/O Latency (ns)		Bandwidth (MiB/s)
	Hash calc	Others	**Data write**	Content-cmp	
NOVA	0.0	84.7	**2275.6**	0.0	1401
Naïve	309.9	1072.5	**585.3**	0.0	1612

**Naïve* means *LD-w/o-P* in our paper

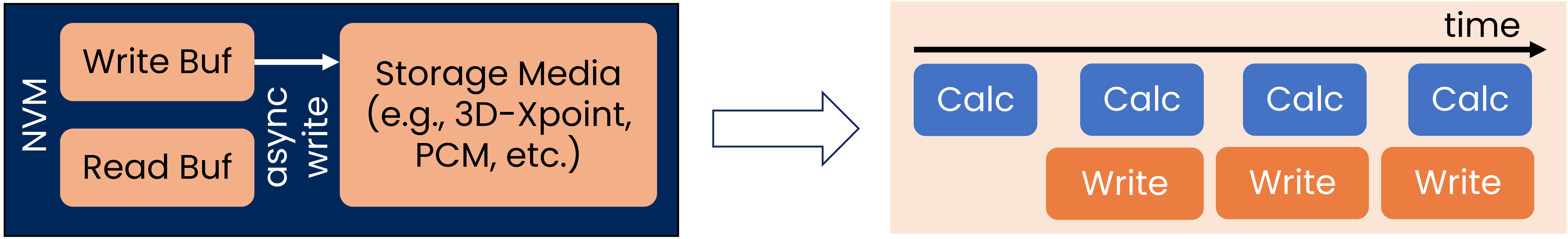
Issue #1. Redundancy Identification Fails to consider I/O Buffers and Read Latency

During the first write

Much less write time!

System	Calc Latency (ns)		I/O Latency (ns)		Bandwidth (MiB/s)
	Hash calc	Others	**Data write**	Content-cmp	
NOVA	0.0	84.7	**2275.6**	0.0	1401
Naïve	**309.9**	**1072.5**	**585.3**	0.0	1612

Not magic! This is caused by async NVM write (with buffers)



I/O and CPU parallelism

Issue #1. Redundancy Identification Fails to consider I/O Buffers and Read Latency

- **Content-comparison can be the bottleneck**
  - NOVA. A state-of-the-art NVM file system
  - *Naïve*. A non-crypto-hash-based Dedup file system
  - Experiment. Write two identical 4GiB files

During the second write

Slower than simply write data

System	Calc Latency (ns)		I/O Latency (ns)		Bandwidth (MiB/s)
	Hash calc	Others	**Data write**	**Content-cmp**	
NOVA	0.0	84.7	**2275.6**	0.0	1401
Naïve	308.0	571.6	0.0	**3263.0**	870

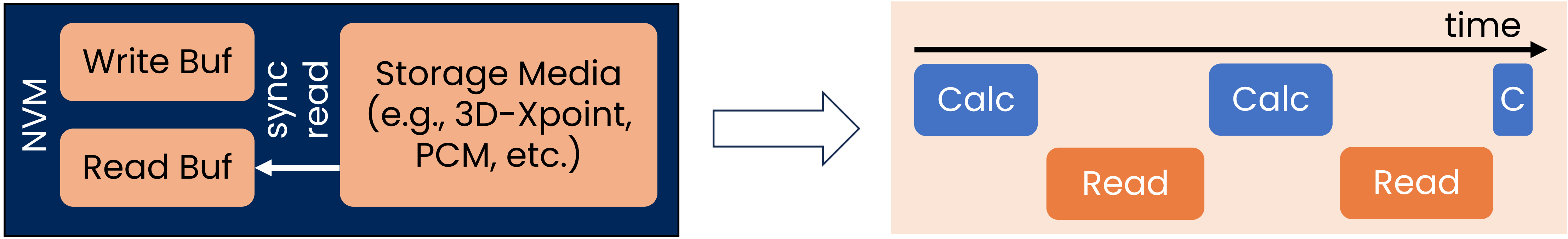
Issue #1. Redundancy Identification Fails to consider I/O Buffers and Read Latency

During the second write

Slower than simply write data

System	Calc Latency (ns)		I/O Latency (ns)		Bandwidth (MiB/s)
	Hash calc	Others	**Data write**	**Content-cmp**	
NOVA	0.0	84.7	**2275.6**	0.0	1401
Naïve	308.0	571.6	0.0	**3263.0**	870

CPU has to wait for the un-cached data to be loaded from NVM



Blocked read.

Issue #1. Redundancy Identification Fails to consider I/O Buffers and Read Latency

During the second write

Slower than simply write data

System	Calc Latency (ns)	I/O Latency (ns)	Bandwidth

Can we achieve async read to hide such high read latency?

CPU has to wait for the un-cached data to be loaded from NVM

Blocked read.

Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

- **Hardware prefetcher should help, but…**
  - NVM is slower than DRAM
  - HW prefetcher is designed for DRAM, prefetching 2 lines ahead
  - HW prefetcher is too conservative for NVM to hide read latency

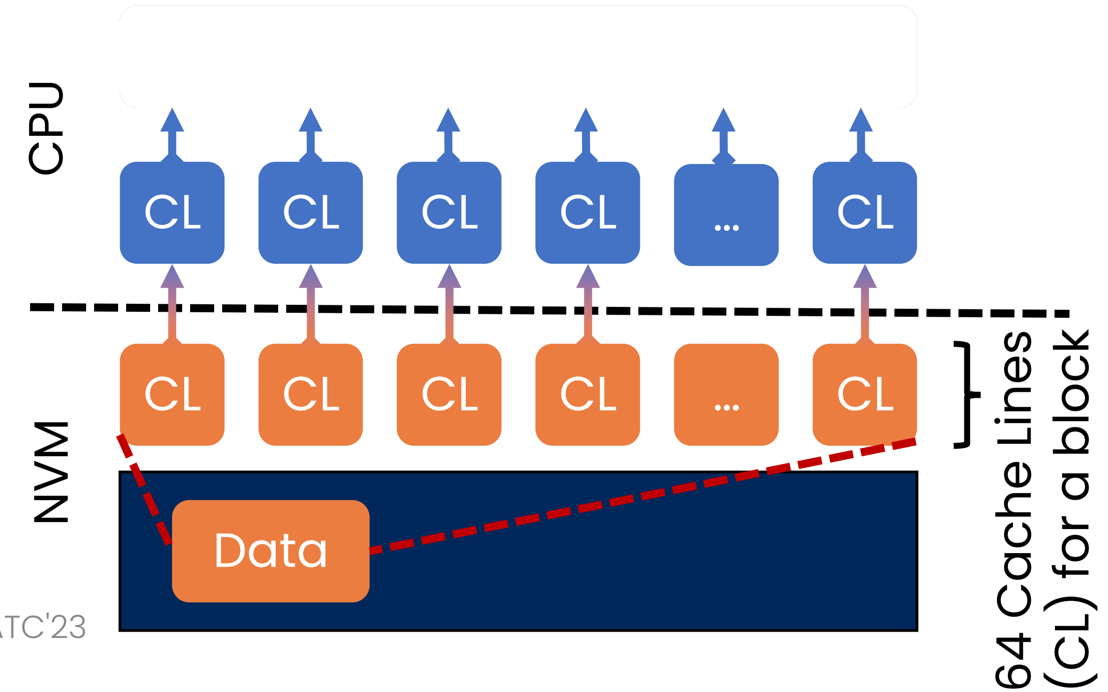
Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

- **Hardware prefetcher should help, but...**
  - NVM is slower than DRAM
  - HW prefetcher is designed for DRAM, prefetching 2 lines ahead
  - HW prefetcher is too conservative for NVM to hide read latency
- **Prefetch more aggressively?**

  Basic Idea

  When content-comparison starts issuing prefetch instruction for every cache line (64 ins in total)

  However, prefetch ins #. that can be handled concurrently is limited

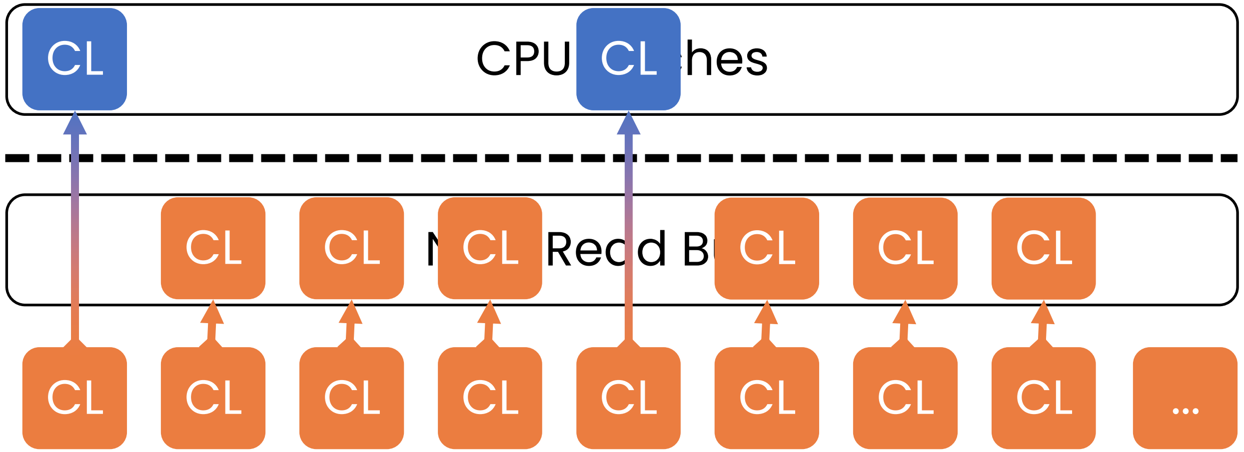


Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

- **Good news: NVM has coarse access granularity!**
  - ✓ NVM typically has a coarser access granularity than cache line
  - ✓ E.g., Optane PMM has a 256 bytes access granularity (XPLine).
  - ✓ No need to issue 64 prefetch ins at first, but only 16!

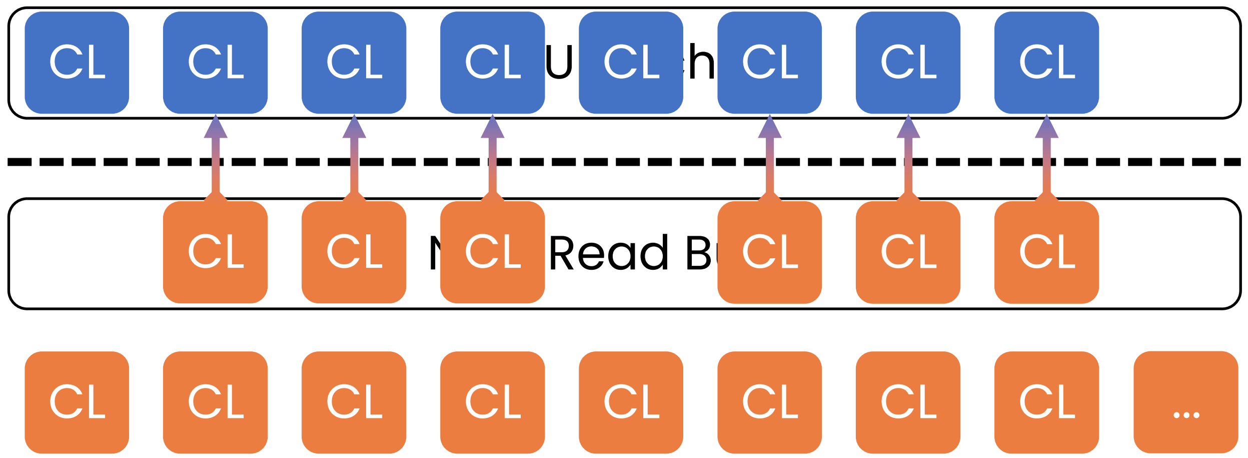
Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

- **Good news: NVM has coarse access granularity!**
  - ✓ NVM typically has a coarser access granularity than cache line
  - ✓ E.g., Optane PMM has a 256 bytes access granularity (XPLine).
  - ✓ No need to issue 64 prefetch ins at first, but only 16!
- **In-Block Prefetch (IBP)**
  1. Issue 16 prefetch ins (prefetch concurrently)



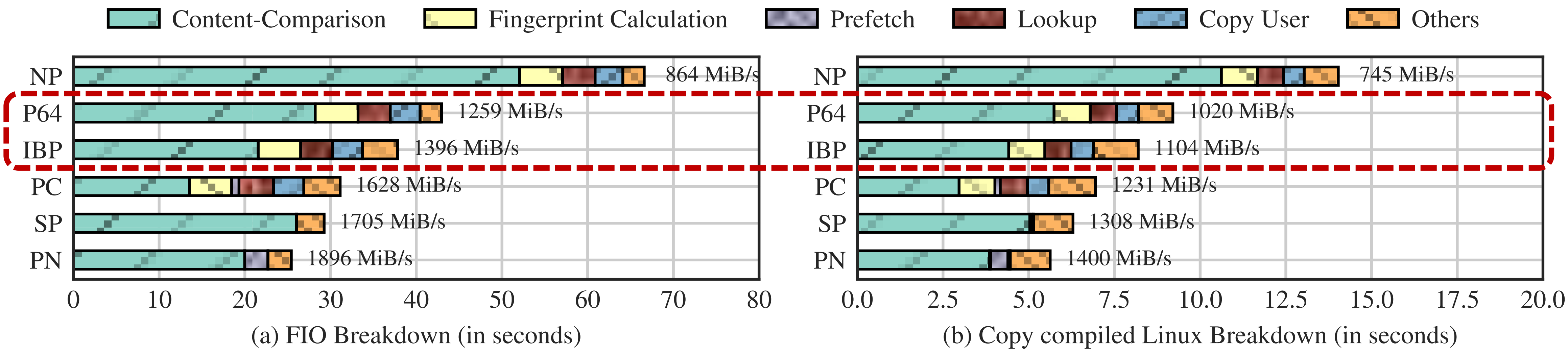
Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

- **Good news: NVM has coarse access granularity!**
  - ✓ NVM typically has a coarser access granularity than cache line
  - ✓ E.g., Optane PMM has a 256 bytes access granularity (XPLine).
  - ✓ No need to issue 64 prefetch ins at first, but only 16!
- **In-Block Prefetch (IBP)**
  1. Issue 16 prefetch ins (prefetch concurrently)
  2. Prefetch the remaining (prefetch from buffer instead of media)



Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

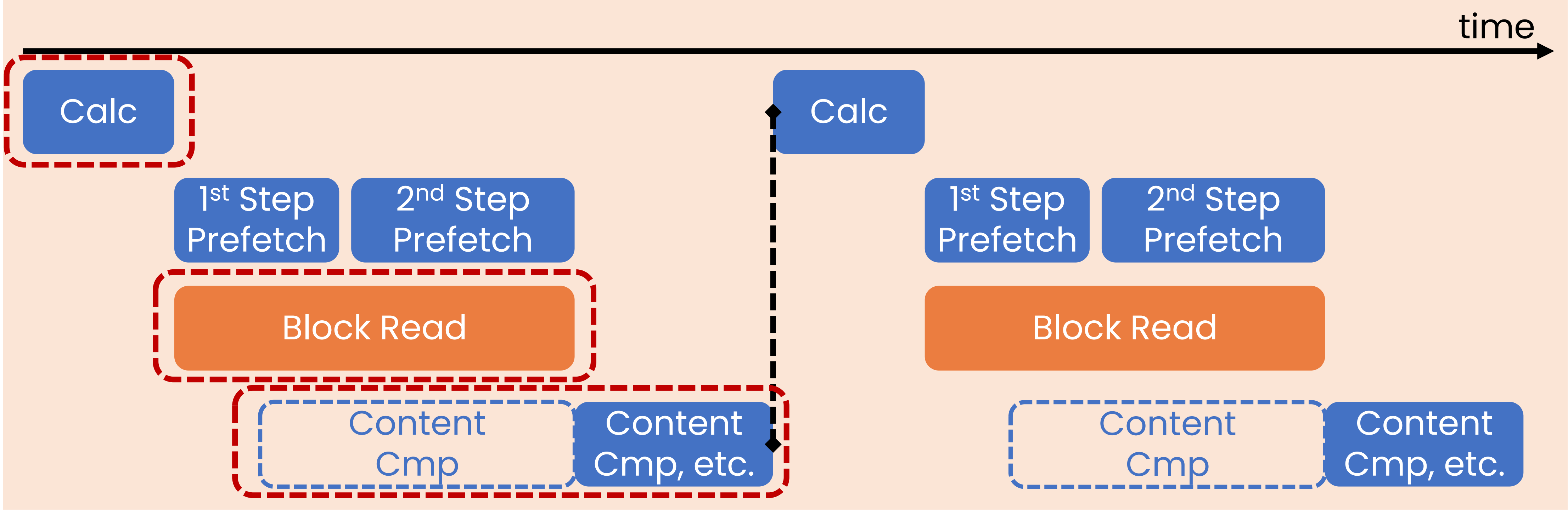
Aggressive Method (P64) vs. IBP



Content-comparison time is dramatically dropped

Solution #1. (1/2) In-Block Prefetch: Using Mem Prefetch to Hide Read Latency

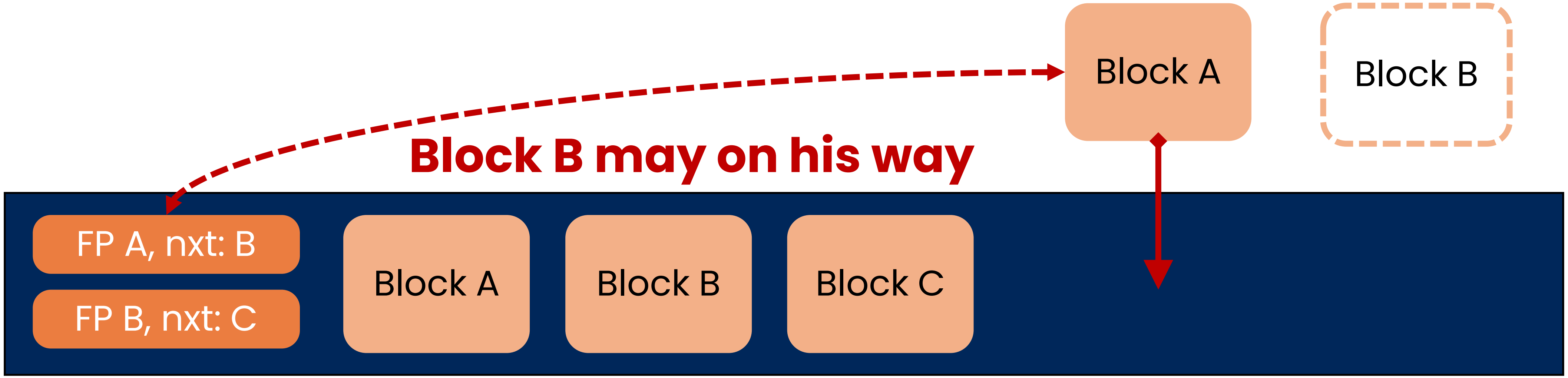
However, IBP cannot exploit the parallelism of CPU (e.g., fingerprint calculation) and I/O



Be parallel like NVM write, How?

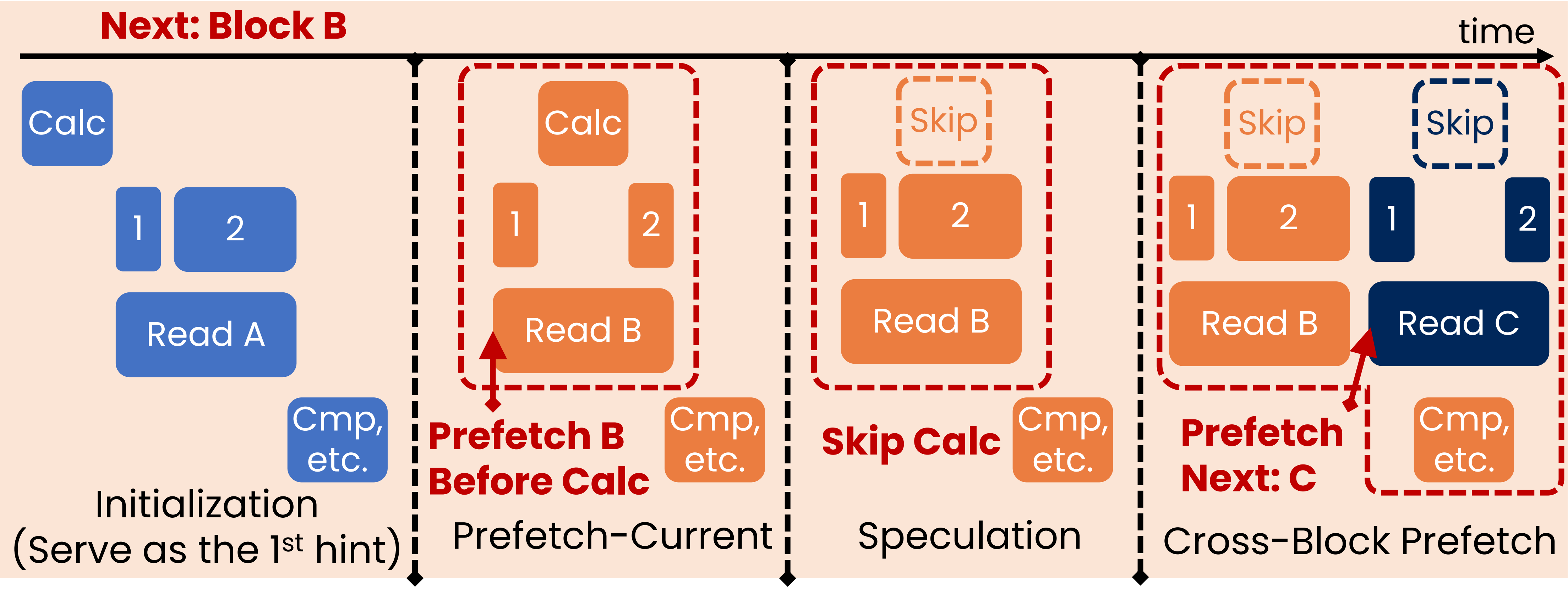
Solution #1. (2/2) Cross-Block Prefetch

- **Key Idea**
  - Speculatively prefetch the **to-be-compared** data block
  - Using a `hint` field in the deduplication metadata entry to record the related information, see our paper for more details.



Solution #1. (2/2) Cross-Block Prefetch

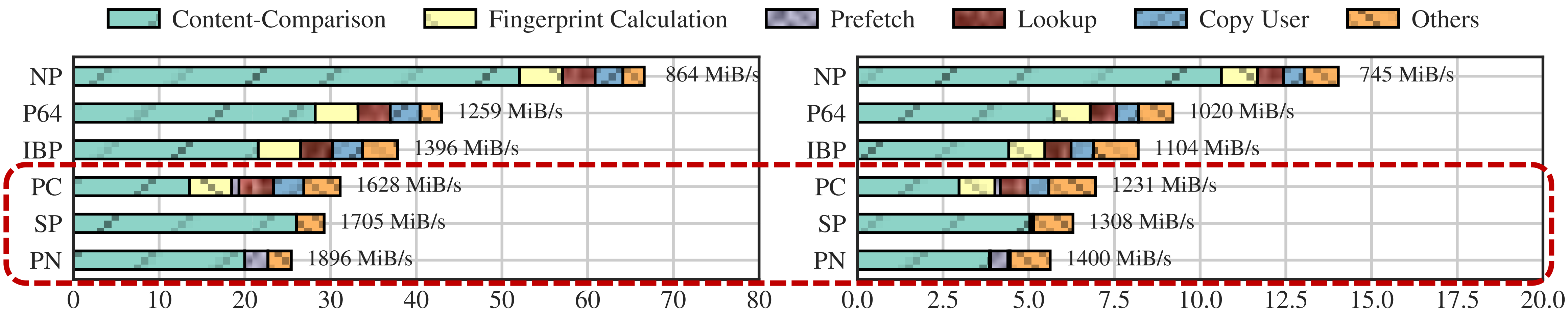
Our three explorations to Cross-Block Prefetch (CBP).



Full Parallelism Achieved!

Solution #1. (2/2) Cross-Block Prefetch

The effectiveness of CBP



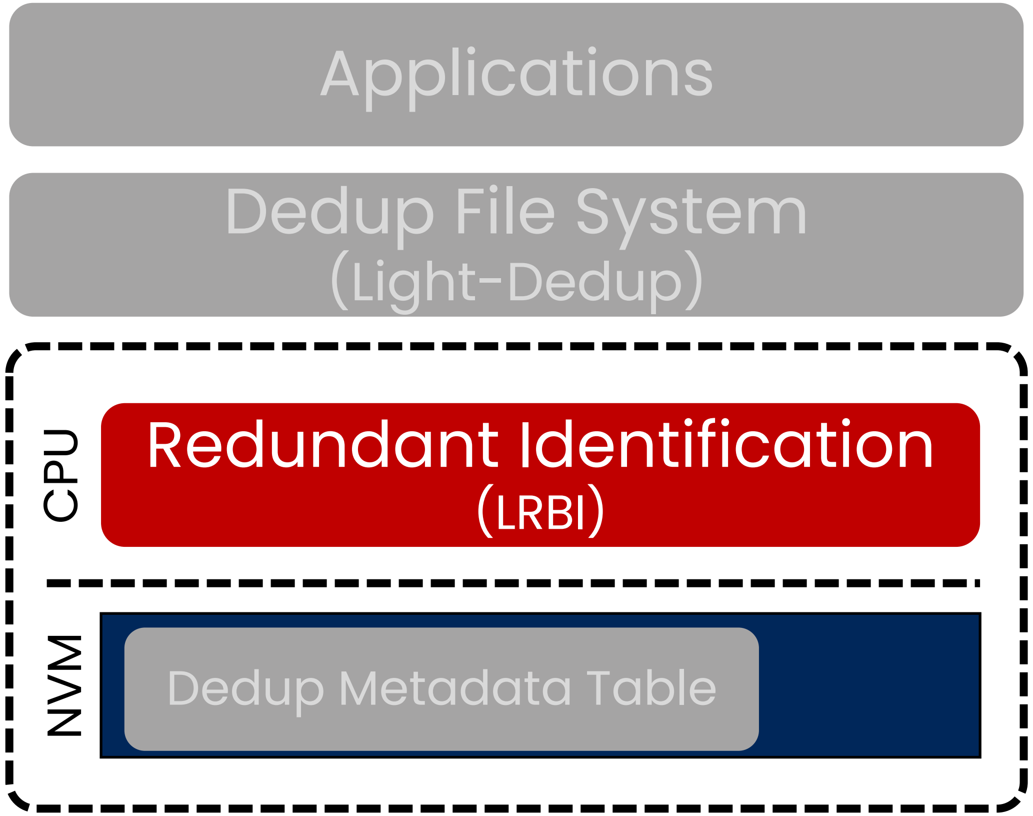
(a) FIO Breakdown (in seconds)

(b) Copy compiled Linux Breakdown (in seconds)

With CBP, CPU calculation is now fully parallel with NVM I/O

Solution #1. Put Together

- **Light-Redundant-Block-Identifier (LRBI)**
  - ✓Non-crypto-hash-based method
  - ✓Speculative-Prefetch-based content-comparison: IBP + CBP



Issue #2. Dedup Metadata Table Fails to consider I/O Amplification

- **Using Dedup metadata table to store**
  - ✓ The mappings between fp to the written data block
  - ✓ Some additional information required by Dedup system, e.g., `hint` for our LRBI
- **Two related questions**
  - ✓ How to efficiently search the entry in the table?
  - ✓ How to manage the layout of Dedup metadata to be NVM friendly?
- **Two existing approaches**
  - **All-in-NVM**. Using static hash table in NVM to achieve fast indexing
  - **Entry-based**. Using in-DRAM structure to quickly index. While using free list to allocate/free entry.

Issue #2. Dedup Metadata Table Fails to consider I/O Amplification

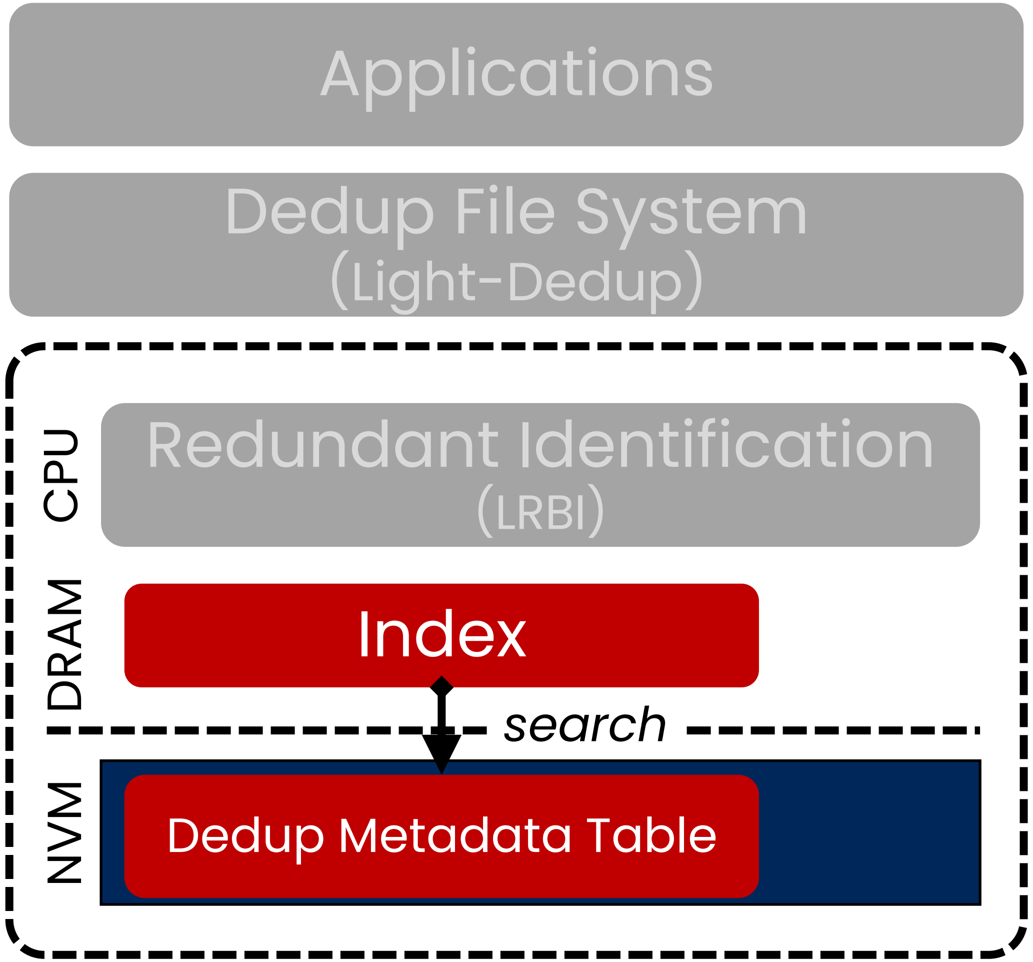
- **However, severe metadata I/O amplification is observed**
  - Reduce NVM's lifetime
  - Reduce deduplication performance

Approaches	First Write		Second Write	
	Meta Read (Bytes/Block)	Meta Write (Bytes/Block)	Meta Read (Bytes/Block)	Meta Write (Bytes/Block)
Ideal	40	40	40	40
All-in-NVM	**726.12**	**293.17**	**528.65**	**259.05**
Entry-based	126.94	79.56	**774.13**	**394.53**

They fail to consider I/O amplification caused by random NVM access

Solution #2. Light-Meta-Table: Managing Dedup Metadata with Locality

- **Key Idea of LMT**
  - ✓ In-DRAM Index. Using in-DRAM index to search for the entry
  - ✓ In-NVM Layout. Allocate meta entries in a coarse region

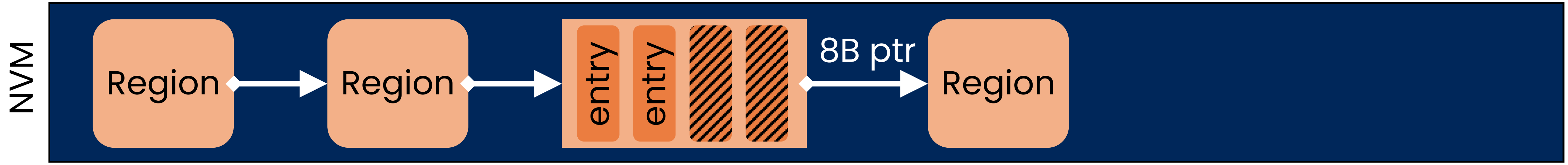


Solution #2. Light-Meta-Table: Managing Dedup Metadata with Locality

- **Index Selection**

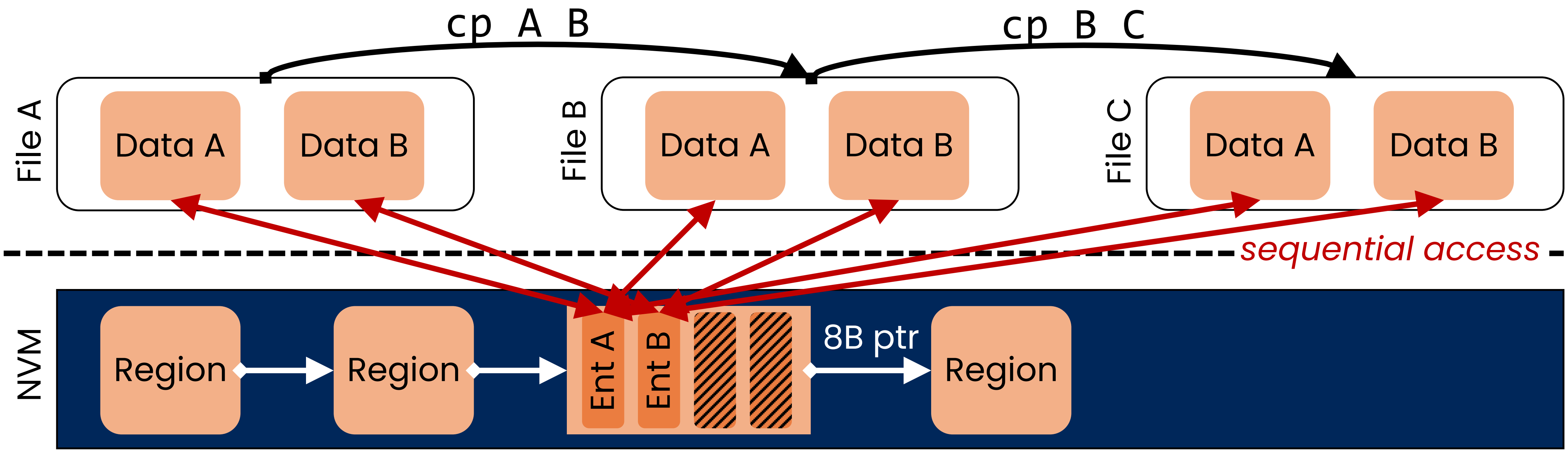
  Using in-DRAM rhashtable (kernel data structure) for its resizable, maturity, and scalability

- **Region-based Layout**
  - Region is in a block size (i.e., 4KiB)
  - Region is linked by a 8 byte pointer to avoid static allocation
  - Region can be **reused** if no less than half entries are freed or empty
  - Trade 1× space consumption (1.56%) for **GC-free**



Solution #2. Light-Meta-Table: Managing Dedup Metadata with Locality

- **How does LMT maintains locality?**
  - Unique write. Entries are allocated almost sequentially in a region
  - Duplicate write. Entries are potentially accessed sequentially, e.g., `cp`



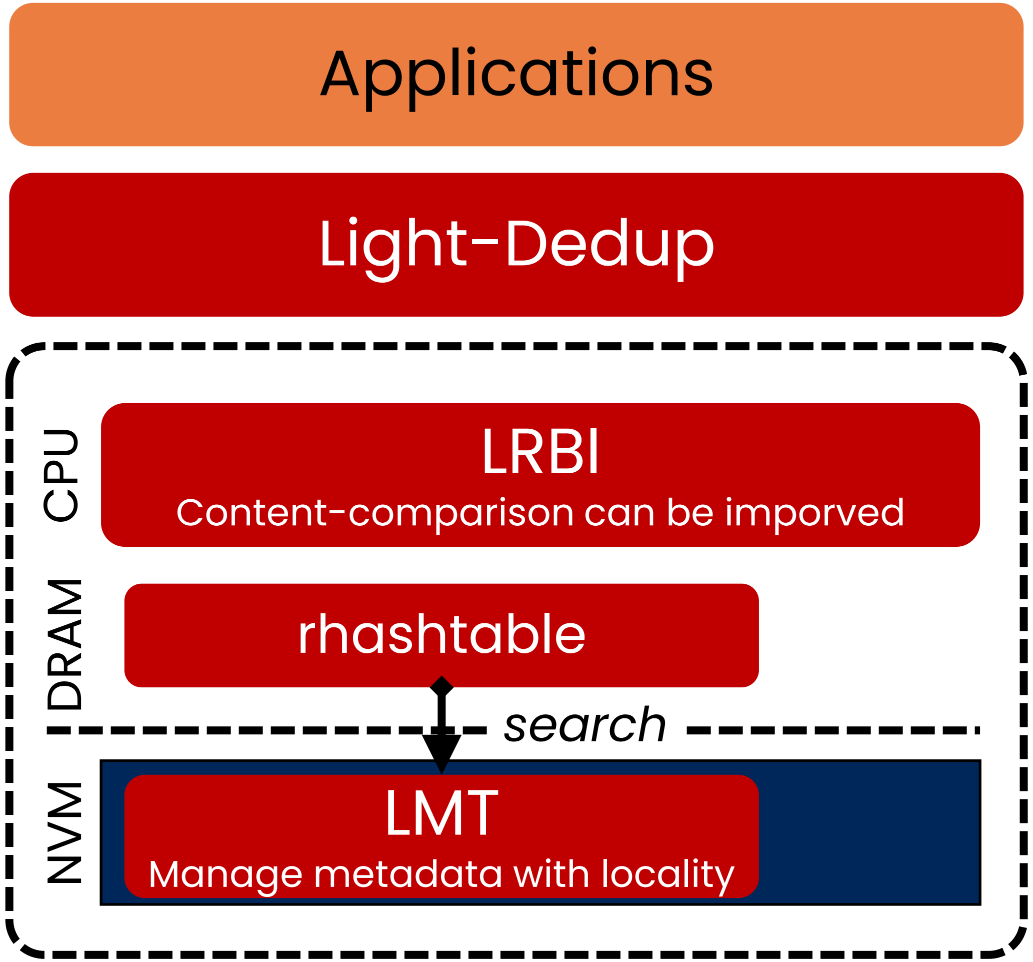
Solution #2. Light-Meta-Table: Managing Dedup Metadata with Locality

- **The effectiveness of LMT (under an aged file system)**
  - Punch holes in file to emulate aging workload
  - Region layout significantly reduces metadata I/O amplification

Dimension	Fresh System (128 GiB)		**Aged System (64 GiB)**	
	LMT (Region Layout)	Entry-based Layout	**LMT (Region Layout)**	Entry-based Layout
Meta Read (Bytes/Block)	116.28 (2.91×)	126.94 (3.17×)	**244.19 (6.1×)**	774.13 (19.35×)
Meta Write (Bytes/Block)	75.75 (1.89×)	79.56 (1.99×)	**137.17 (3.43×)**	394.54 (9.86×)
Throughput (MiB/s)	1747.5	1690.6	**1336.72**	1197.76

Solution #1 & #2. Light-Dedup

- **Light-Redundant-Block-Identifier (LRBI)**
  - ✓Non-crypto-hash-based method (NVM's read/write asymmetry)
  - ✓Speculative-Prefetch-based content-comparison: IBP + CBP
- **Light-Meta-Table (LMT)**
  - ✓Using region-based layout to maintain good locality



Solution #1 & #2. Light-Dedup

- **Light-Redundant-Block-Identifier (LRBI)**
  - ✓Non-crypto-hash-based method (NVM's read/write asymmetry)
  - ✓Speculative-Prefetch-based content-comparison: IBP + CBP
- **Light-Meta-Table (LMT)**
  - ✓Using region-based layout to maintain good locality

Recall NVM features

- ★ **Read/Write Asymmetry**
- ★ **I/O with Buffers**
- ★ **Long Media Read Latency**
- ★ **Coase Access Granularity**
- ★ **Memory Interface**

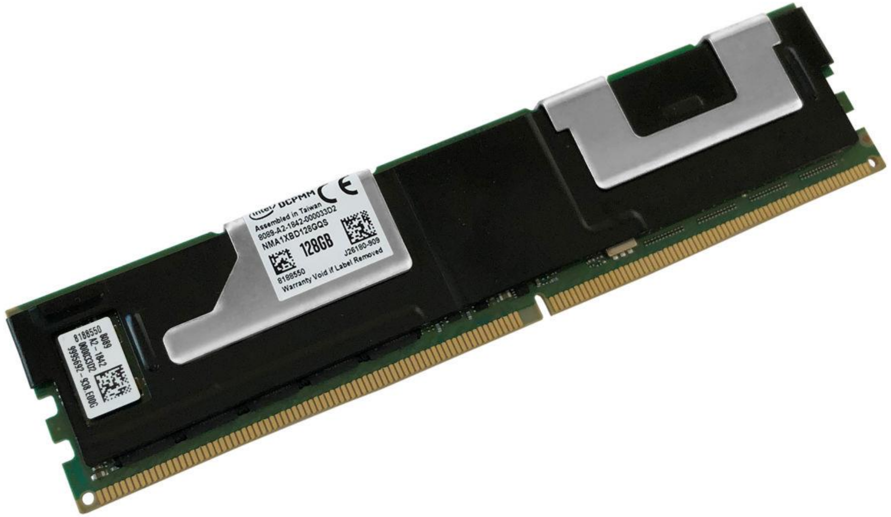
All the features are considered!

More Details in Our Paper

- **The fields of the deduplication entry**
- **When to trust hint**
- **Detailed entry management of LMT**
- **Crash consistency**
- **Portability of Light-Dedup**

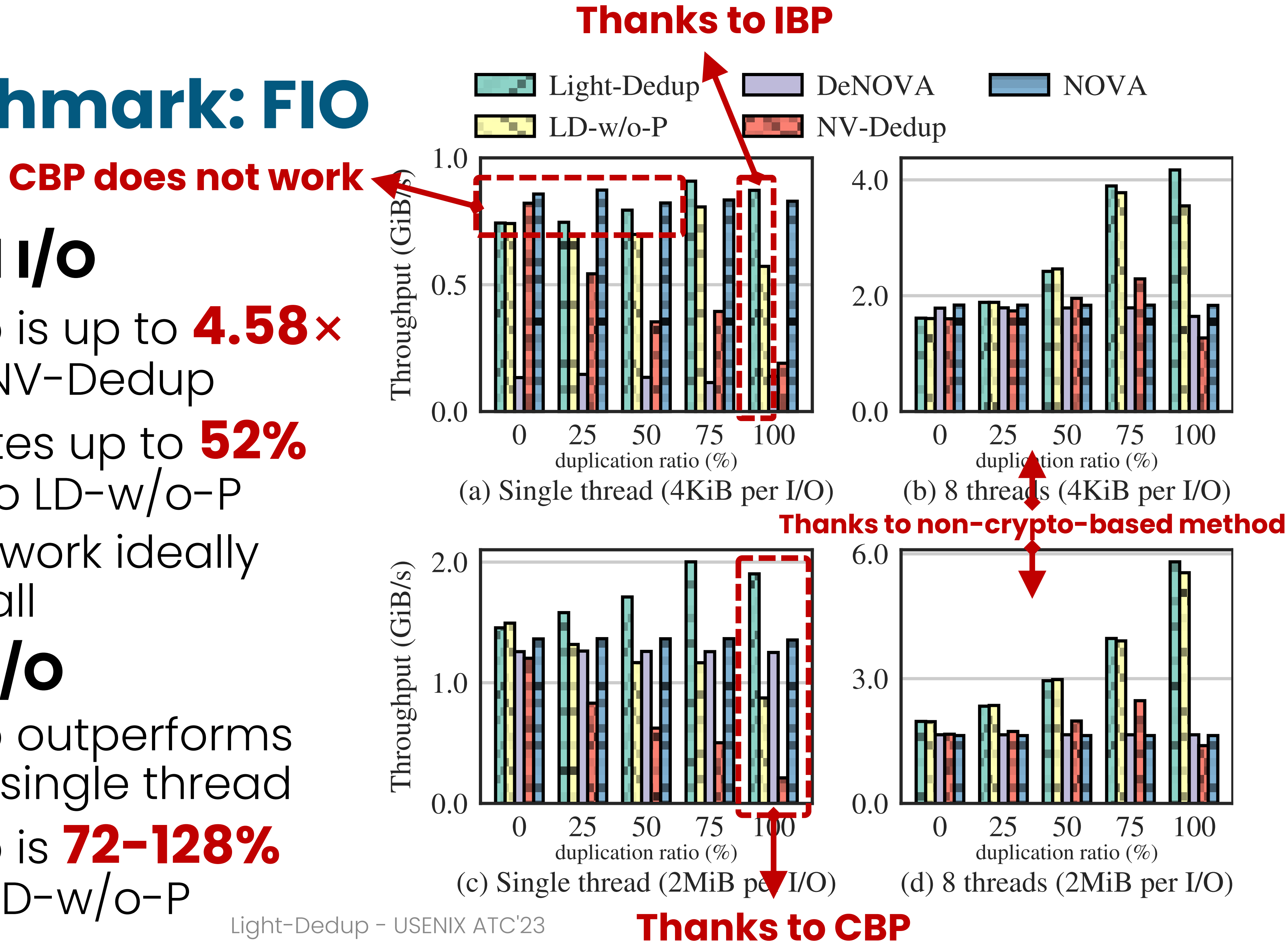
Light-Dedup Evaluation: Setup

- **Linux Kernel 5.1.0 (same as NOVA)**
- **Intel Xeon Gold 5218 CPU @ 2.3GHz**
- **256 GiB Optane DCPMM**
- **128 GiB DRAM**



Microbenchmark: FIO

- **Block-based I/O**
  - ✓Light-Dedup is up to **4.58×** faster than NV-Dedup
  - ✓IBP contributes up to **52%** compared to LD-w/o-P
  - ✓CBP cannot work ideally across syscall
- **Continuous I/O**
  - ✓Light-Dedup outperforms NOVA in the single thread
  - ✓Light-Dedup is **72-128%** faster than LD-w/o-P

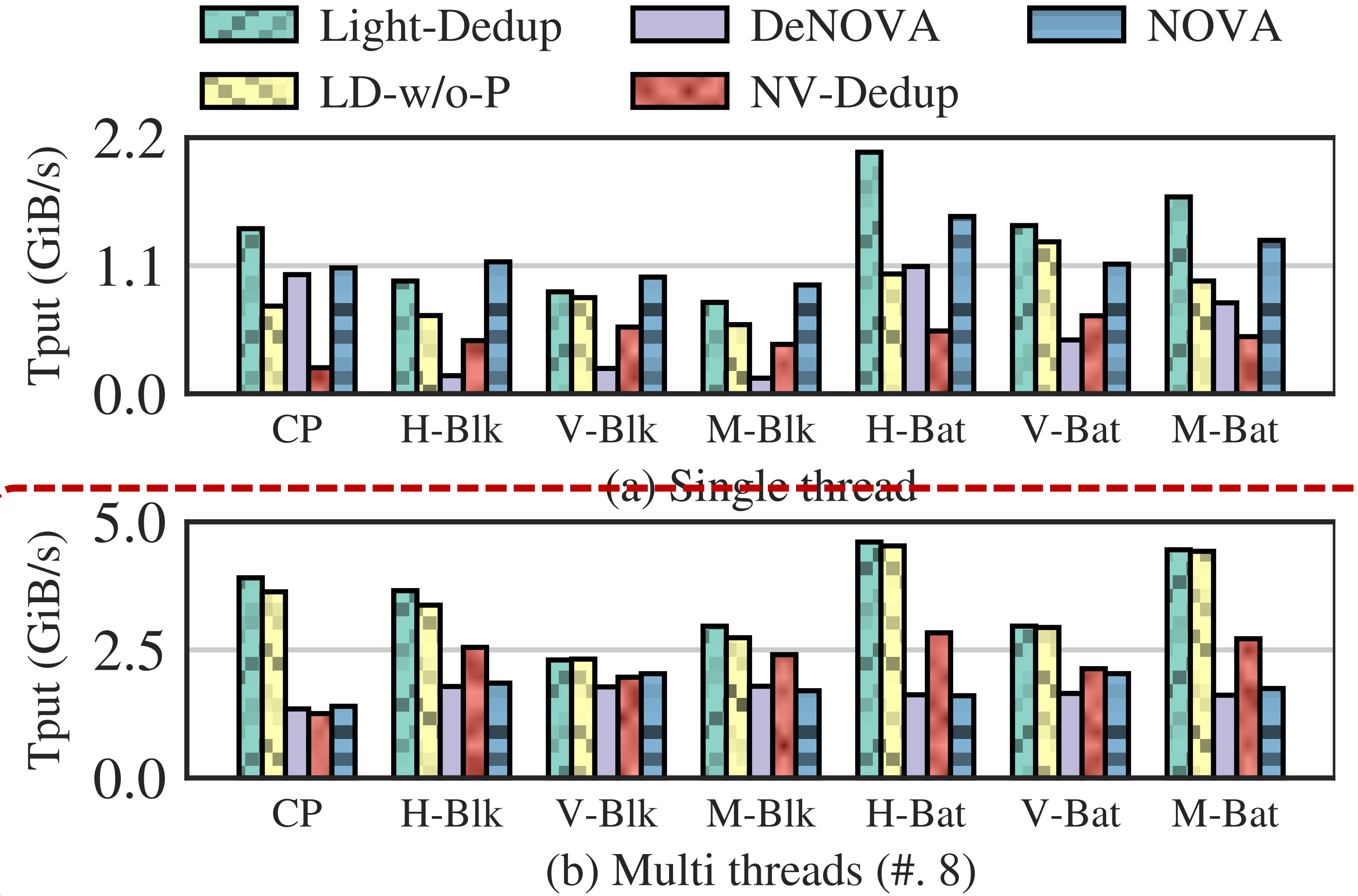


Real-world Scenarios

- **Evaluated read-world workload**
  - Using trace-replayer to replay the trace, which can **batch** the data blocks

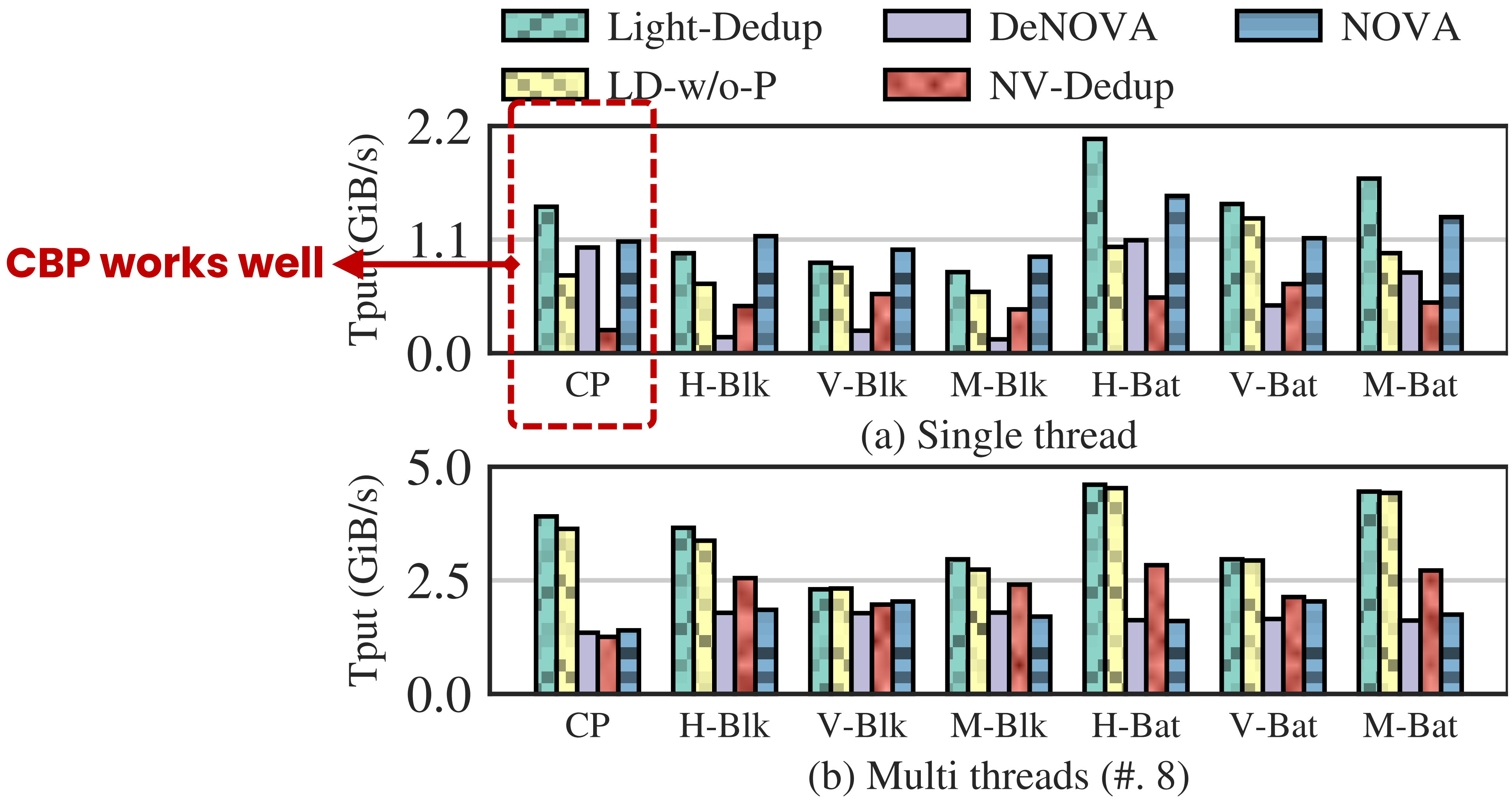
Workload	Scenario	Total I/O	Write Prop.	**Dup Ratio**	Granularity
Copy	Copy Compiled Linux Kernel	13.85 GiB	100%	**100%**	2 MiB
Homes (Trace)	Our OS Lab	63.52 GiB	100%	**84%**	4KiB for Blk Max 2MiB for Bat
WebVMs (Trace)	Two Web Servers (FIU Trace)	54.53 GiB	78%	**47%**	4KiB for Blk Max 2MiB for Bat
Mails (Trace)	An Email Server (FIU Trace)	173.27 GiB	91%	**96%**	4KiB for Blk Max 2MiB for Bat

Real-world Scenarios



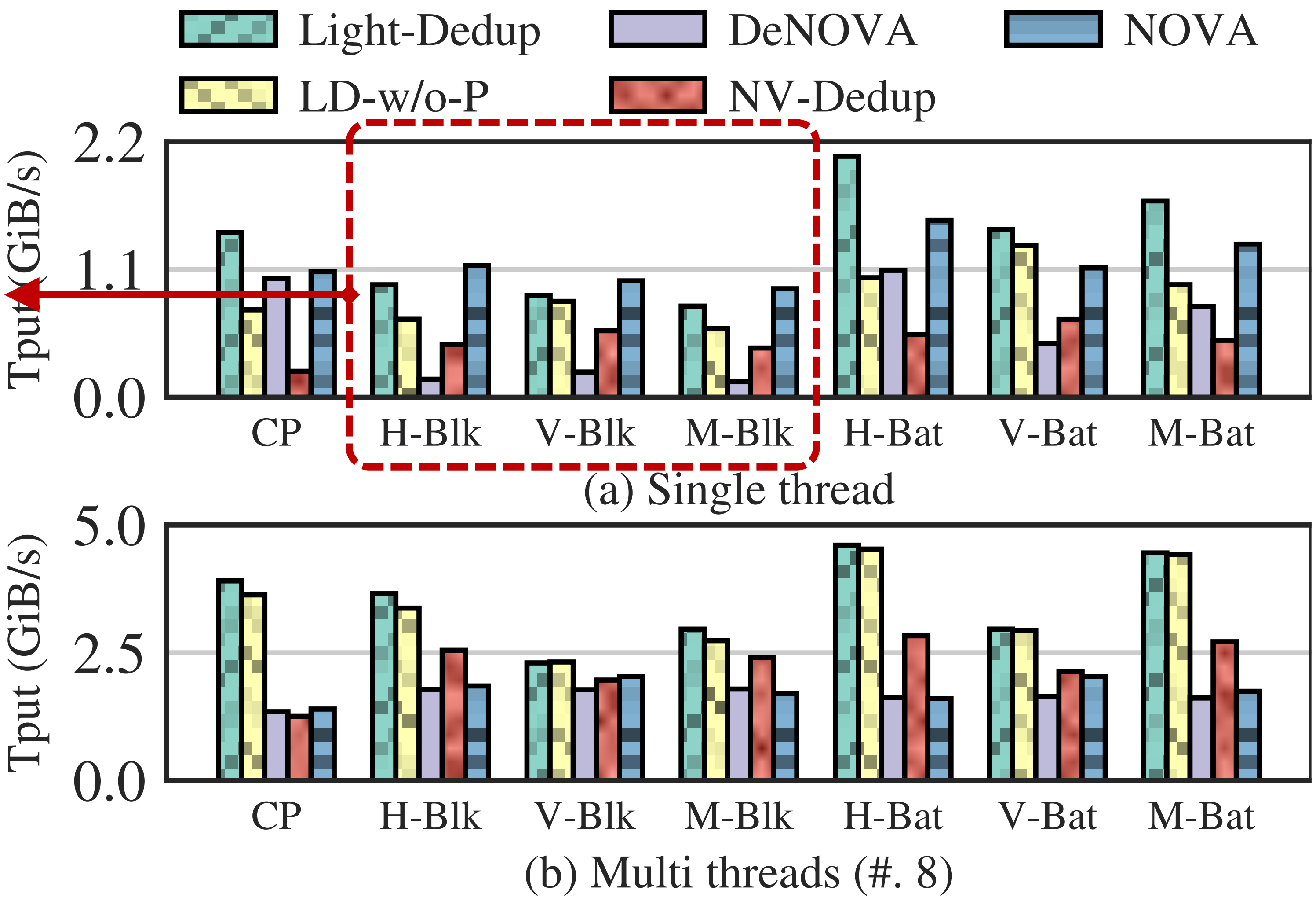
Thanks to non-crypto-based method (similar to FIO)

Real-world Scenarios

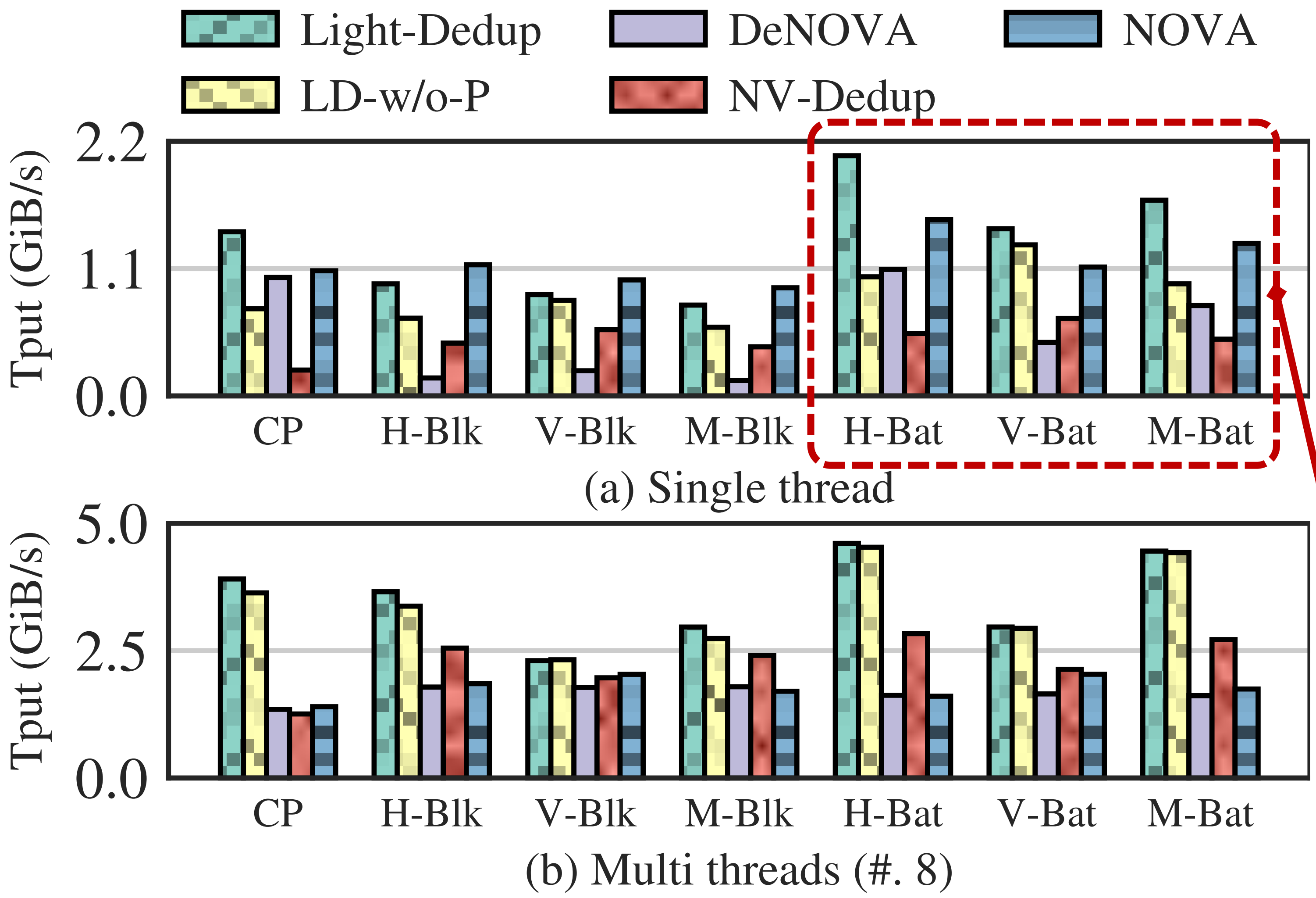


Real-world Scenarios

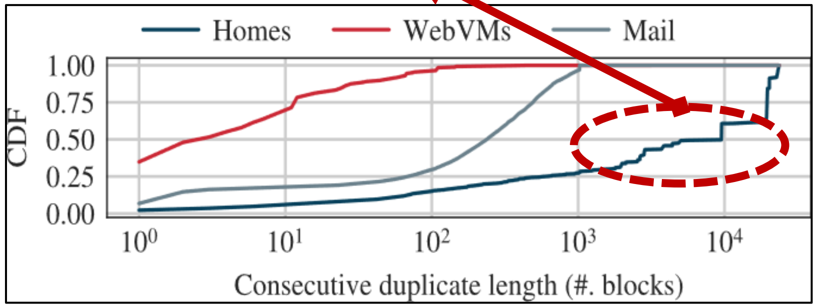
Consistent with FIO workload. IBP contributes here



Real-world Scenarios



Best consecutive duplication



- **CBP contributes to batched trace a lot**
- **Light-Dedup performs best under Homes-Bat**

Conclusion

- **Deduplication can largely reduce NVM's cost**
- **Existing approaches fail to fully exploit NVM's I/O features**
  - I/O with buffers, long read latency, memory interface, and coarse access granularity
- **We propose Light-Dedup to fully exploit these features**
  - ✓LRBI. Speculative-prefetch-accelerated content-comparison
  - ✓LMT. Region-based layout and in-DRAM rhashtable
  - ✓Significant speedup against SOTA methods with low meta I/O amp.

github.com/Light-Dedup/

Check out our paper for more details!

Thanks & QA