Toward I/O-Efficient Protection Against Silent Data Corruptions in RAID Arrays

Mingqiang Li and **Patrick P. C. Lee**

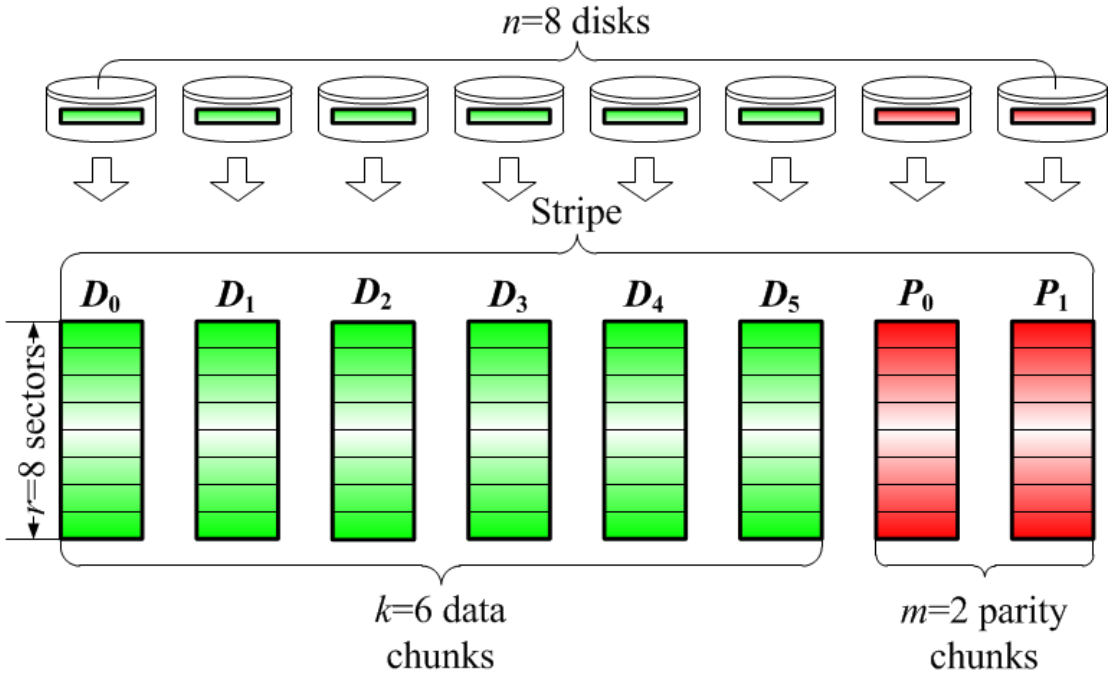
The Chinese University of Hong Kong

MSST ’14

[Patterson et al., SIGMOD ’88]

RAID

- **RAID** is known to protect data against disk failures and latent sector errors
  - How it works? Encodes *k* data chunks into *m* parity chunks, such that the *k* data chunks can be recovered from any *k* out of *n=k+m* chunks



Silent Data Corruptions

Silent data corruptions:

- Data is stale or corrupted without indication from disk drives → cannot be detected by RAID
- Generated due to firmware or hardware bugs or malfunctions on the read/write paths
- More dangerous than disk failures and latent sector errors

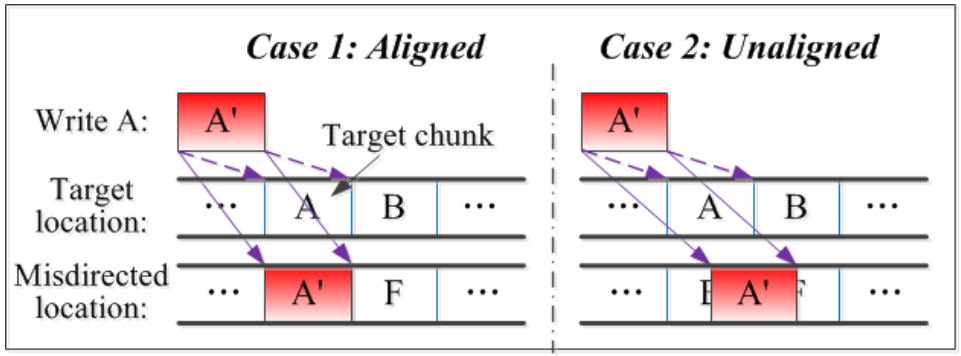
[Kelemen, LCSC ’07; Bairavasundaram et al., FAST ’08; Hafner et al., IBM JRD 2008]

Silent Data Corruptions

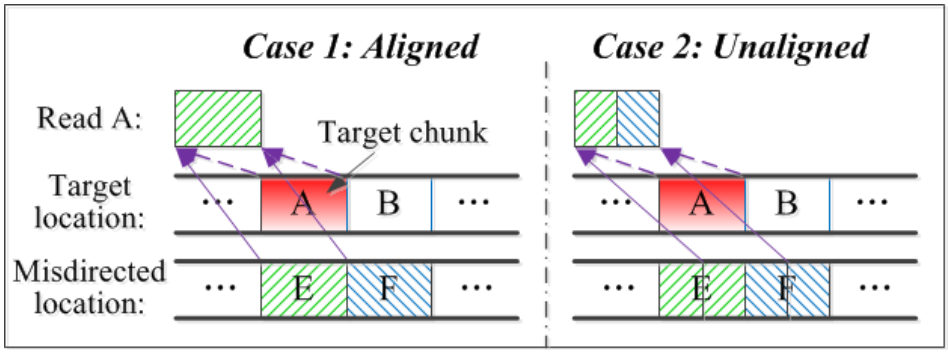
- Lost write:

- Torn write:

- Misdirected writes/reads:



(a) Misdirected writes



(b) Misdirected reads

Silent Data Corruptions

Consequences:

- User read:
  - Corrupted data propagated to upper layers
- User write:
  - Parity pollution
- Data reconstruction
  - Corruptions of surviving chunks propagated to reconstructed chunks

Integrity Protection

- Protection against silent data corruptions:
  - Extend RAID layer with integrity protection, which adds **integrity metadata** for detection
  - Recovery is done by RAID layer
- Goals:
  - All types of silent data corruptions should be detected
  - Reduce computational and I/O overheads of generating and storing integrity metadata
  - Reduce computational and I/O overheads of detecting silent data corruptions

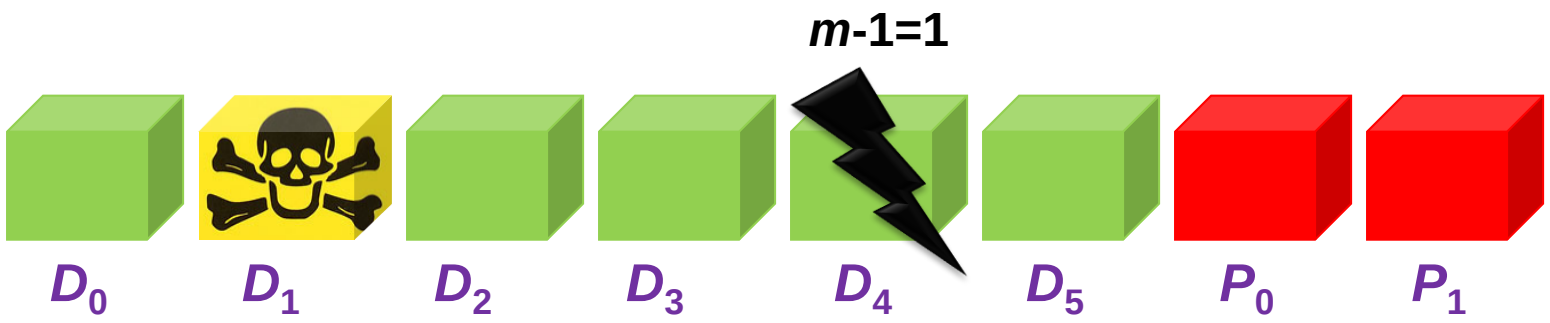
Our Contributions

- A taxonomy study of existing integrity primitives on I/O performance and detection capabilities
- An integrity checking model
- Two I/O-efficient integrity protection schemes with complementary performance gains
- Extensive trace-driven evaluations

Assumptions

- At most one silently corrupted chunk within a stripe
- If a stripe contains a silently corrupted chunk, the stripe has no more than m-1 failed chunks due to disk failures or latent sector errors

Otherwise, higher-level RAID is needed!

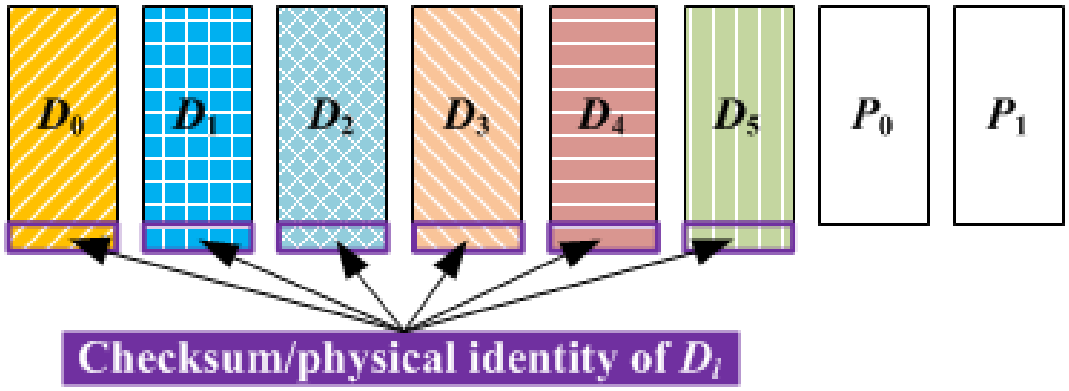


How RAID Handles Writes?

- **Full-stripe writes**:
  - Parity chunks are computed directly from data chunks to be written chunks (no disk reads needed)
- **Partial-stripe writes**:
  - **RMW** (Read-modify-writes) → for small writes
    - Read all touched data chunks and all parity chunks
    - Compute the data changes and the parity chunks
    - Write all touched data chunks and parity chunks
  - **RCW** (Reconstruct-writes) → for large writes
    - Read all untouched data chunks
    - Compute the parity chunks
    - Write all touched data chunks and parity chunks

Existing Integrity Primitives

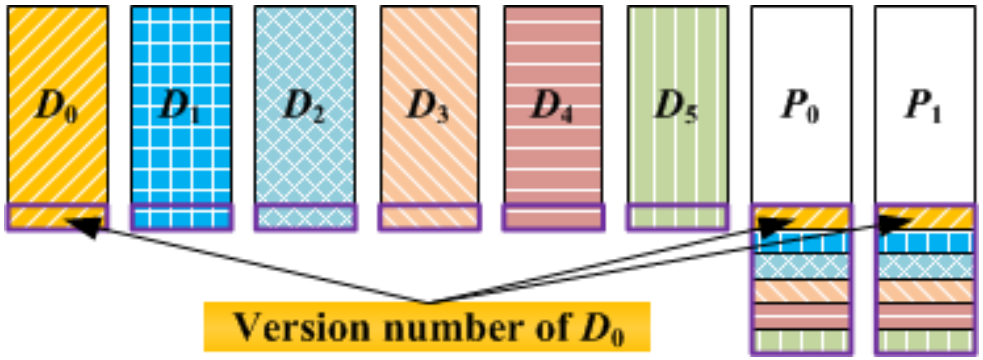
- Self-checksumming / Physical identity [Krioukov et al., FAST '08]



- Data and metadata are read in a single disk I/O
- Inconsistency implies data corruption
- Cannot detect stale or overwritten data

Existing Integrity Primitives

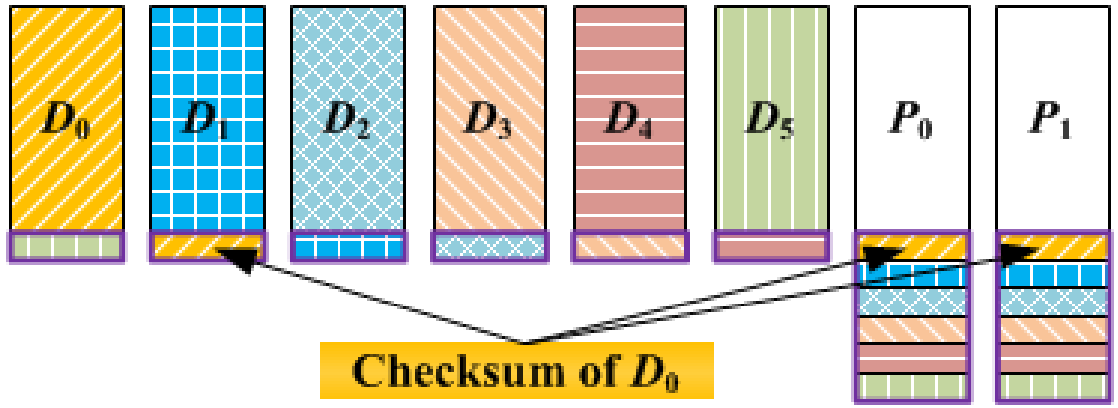
- Version Mirroring [Krioukov et al., FAST '08]



- Keep a version number in the same data chunk and *m* parity chunks
- Can detect lost writes
- Cannot detect corruptions

Existing Integrity Primitives

- Checksum Mirroring [Hafner et al., IBM JRD 2008]



- Keep a checksum in the neighboring data chunk (***buddy***) and *m* parity chunks
- Can detect all silent data corruptions
- High I/O overhead on checksum updates

Comparisons

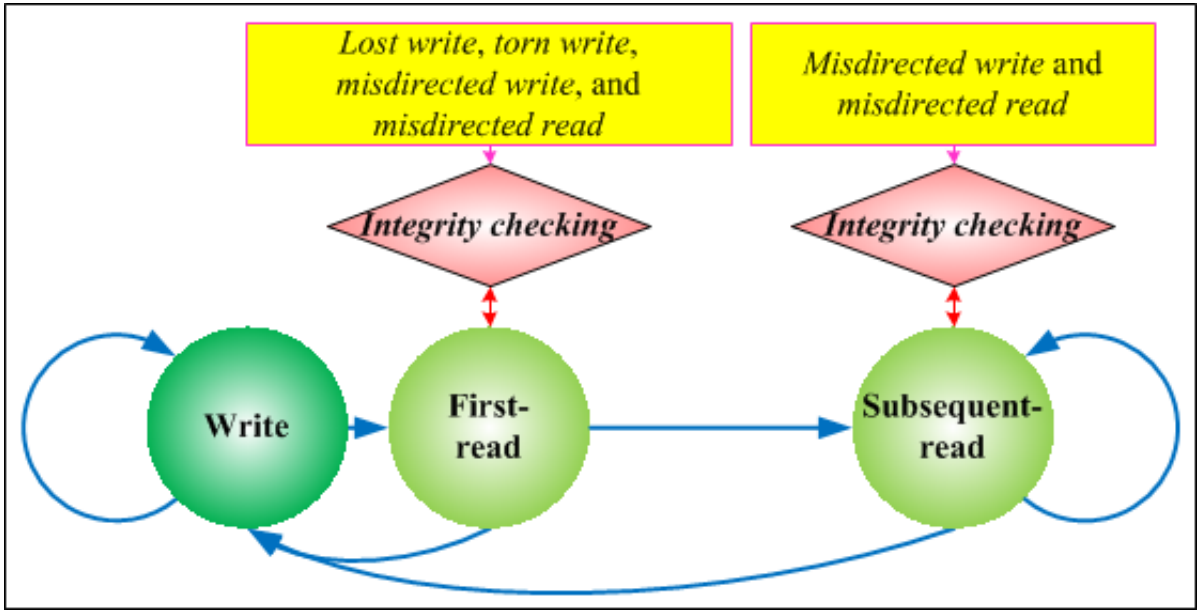
No additional I/O overhead

Integrity Primitives		Detection Capabilities for Different Types of Silent Data Corruptions						
		Lost write	Torn write	Misdirected write			Misdirected read	
				Aligned	Unaligned		Aligned	Un-aligned
					Front-part	End-part		
Self-checksumming	*Self-checking*		✓		✓	✓		✓
Physical identity				✓		✓	✓	✓
Version mirroring	*Cross-checking*	✓						
Checksum mirroring		✓	✓	✓	✓	✓	✓	✓

Additional I/O overhead

Question: ***How to integrate integrity primitives into I/O-efficient integrity protection schemes?***

Integrity Checking Model



- Two types of disk reads:
  - **First read**: sees all types of silent data corruptions
  - **Subsequent reads**: see a subset of types of silent data corruptions
- Observation: A simpler and lower-overhead integrity checking mechanism is possible for subsequent-reads

Checking Subsequent-Reads

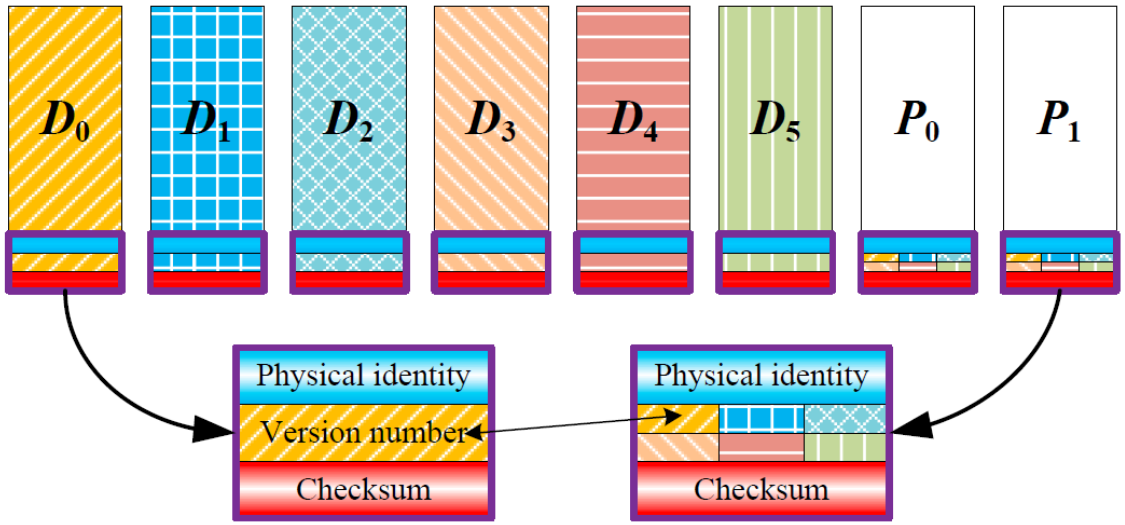
Seen by subsequent-reads

Integrity Primitives		Detection Capabilities for Different Types of Silent Data Corruptions							Integrity Protection Schemes		
		Lost write	Torn write	Misdirected write			Misdirected read		PURE (♢)	HYBRID-1 (♣)	HYBRID-2 (♠)
				Aligned	Unaligned		Aligned	Un-aligned			
					Front-part	End-part					
Self-checksumming	*Self-checking*		✓		✓	✓		✓		♣	♠
Physical identity	*Self-checking*			✓		✓	✓	✓		♣	♠
Version mirroring	*Cross-checking*	✓								♣	
Checksum mirroring	*Cross-checking*	✓	✓	✓	✓	✓	✓	✓	♢		♠

No additional I/O overhead

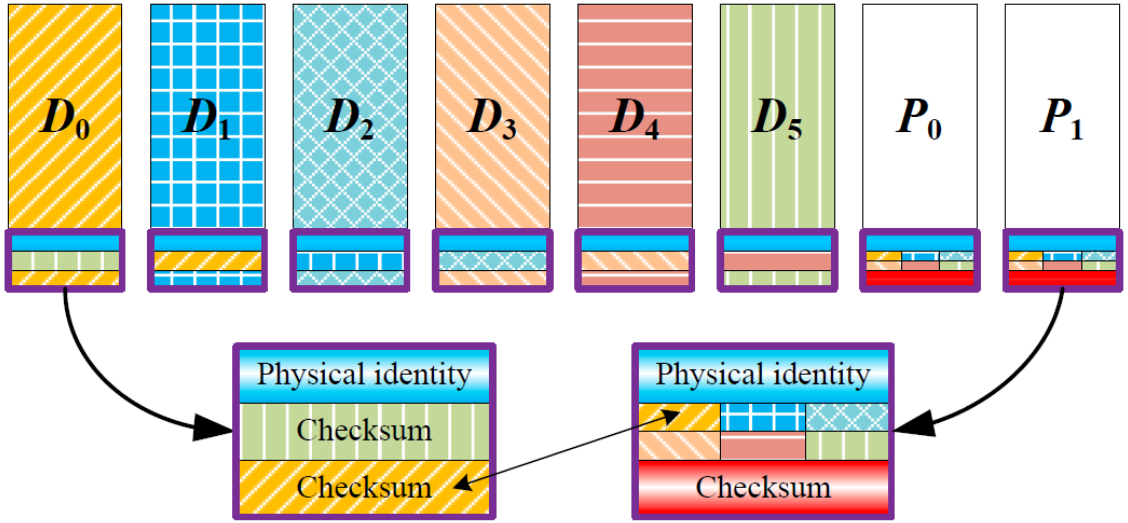
- Subsequent-reads can be checked by **self-checksumming and physical identity without additional I/Os**
- Integrity protection schemes to consider:
  - PURE (checksum mirroring only), HYBRID-1, and HYBRID-2

Integrity Protection Schemes



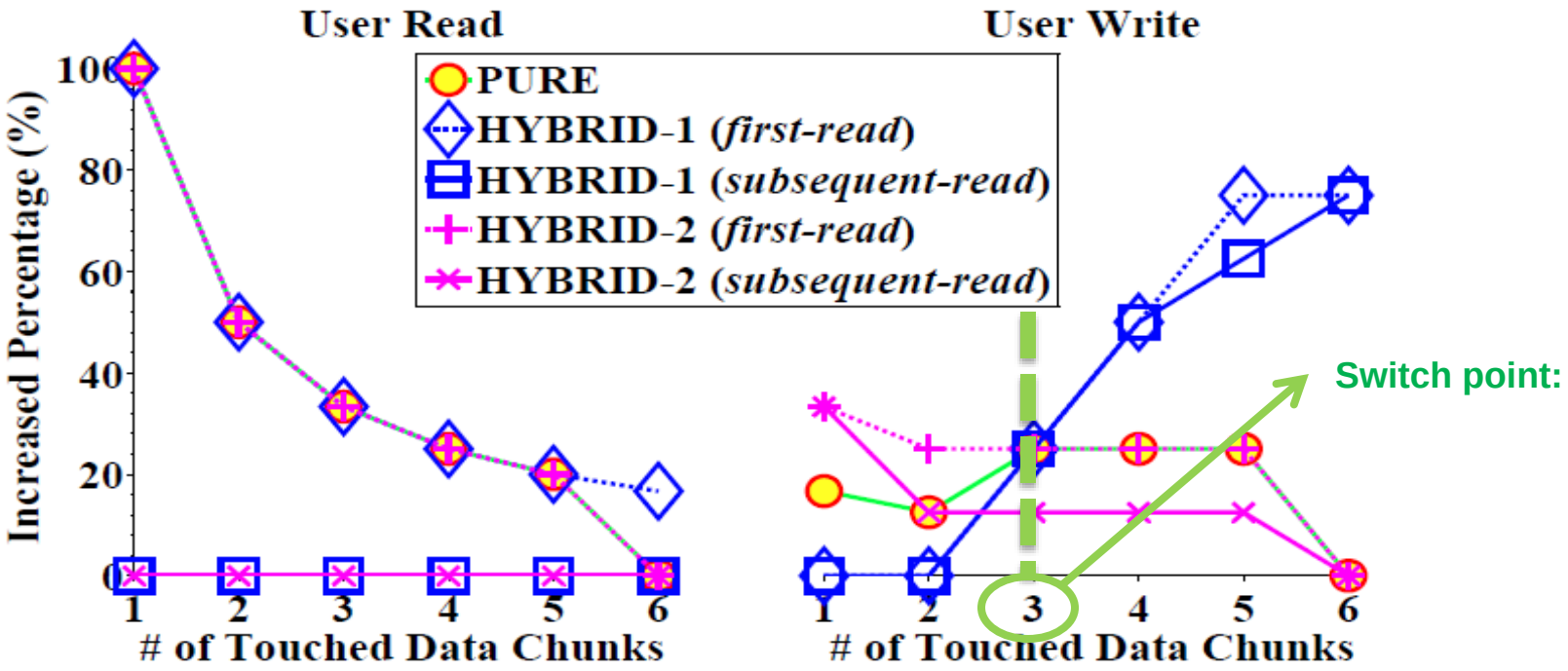
- Hybrid-1
  - Physical identity + self-checksumming + **version mirroring**
  - A variant of the scheme in [Krioukov et al., FAST '08]

Integrity Protection Schemes



- Hybrid-2
  - Physical identity + self-checksumming + **checksum mirroring**
  - A **NEW** scheme

Additional I/O Overhead for a Single User Read/Write



Both Hybrid-1 and Hybrid-2 outperform Pure in subsequent-reads

Hybrid-1 and Hybrid-2 provide complementary I/O advantages for different write sizes

Choosing the Right Scheme

- If $\frac{\overline{S}_{write}}{S_{chunk}} \leq \left\lceil \frac{n+1}{2} \right\rceil - m,$ choose Hybrid-1
- If $\frac{\overline{S}_{write}}{S_{chunk}} > \left\lceil \frac{n+1}{2} \right\rceil - m,$ choose Hybrid-2

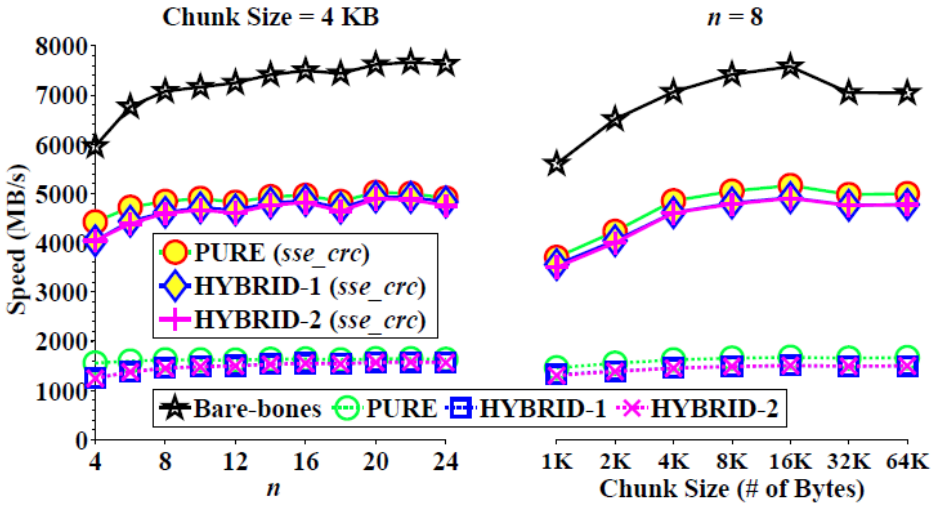
- $\overline{S}_{write}$ = average write size of a workload (estimated through measurements)
- S_{chunk} = RAID chunk size
- The chosen scheme is configured in the RAID layer (offline) during initialization

Evaluation

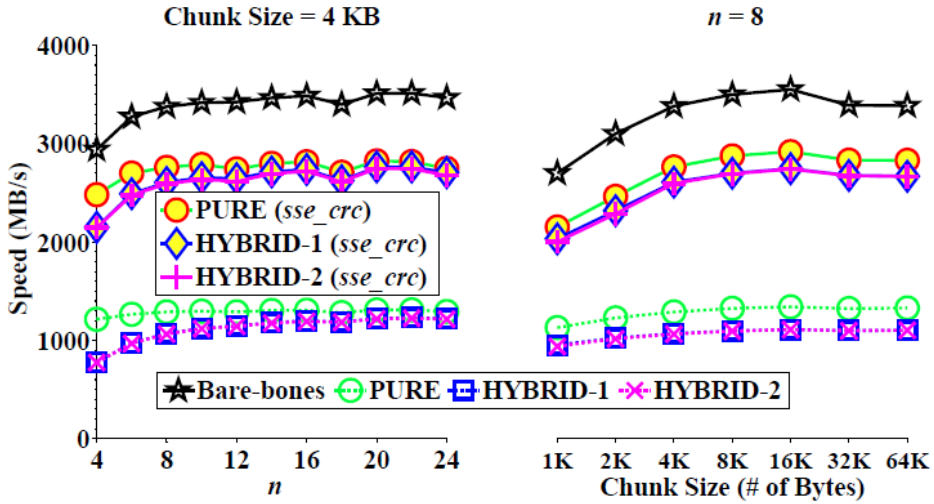
- Computational overhead for calculating integrity metadata
- I/O overhead for updating and checking integrity metadata
- Effectiveness of choosing the right scheme

Computational Overhead

- Implementation:
  - GF-Complete [Plank et al., FAST'13] and Crcutil libraries
- Testbed:
  - Intel Xeon E5530 CPU @ 2.4GHz with SSE4.2
- Overall results:
  - ~4GB/s for RAID-5
  - ~2.5GB/s for RAID-6
- **RAID performance is bottlenecked by disk I/Os, rather than CPU**



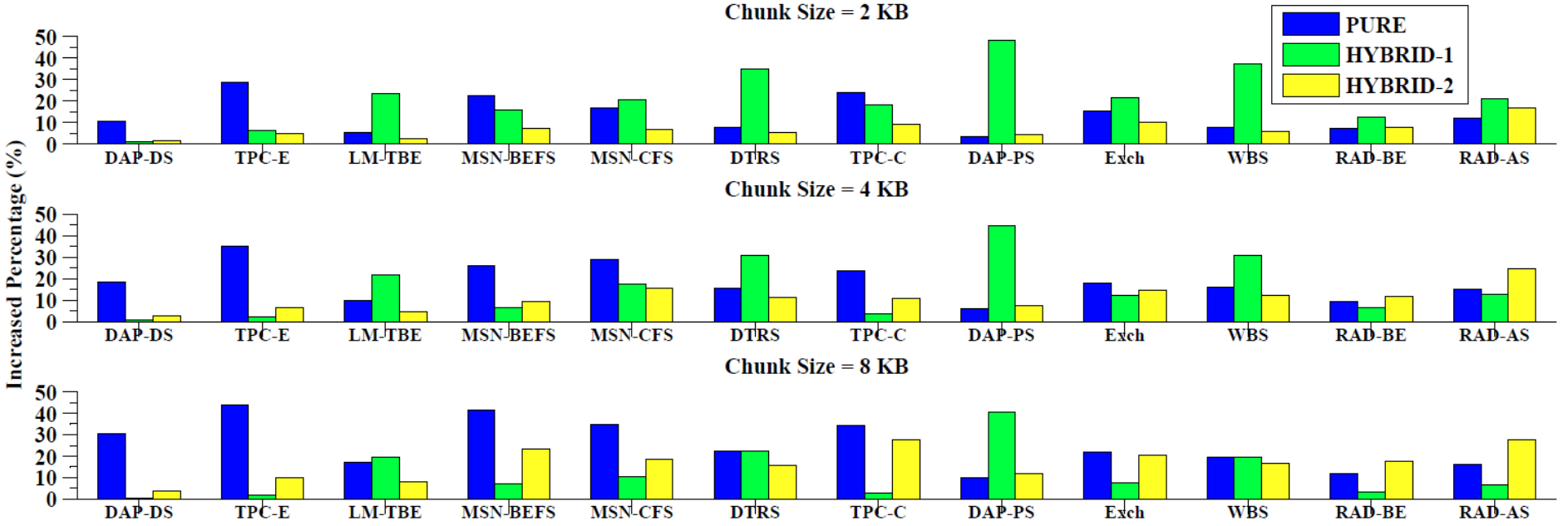
(a) RAID-5



(b) RAID-6

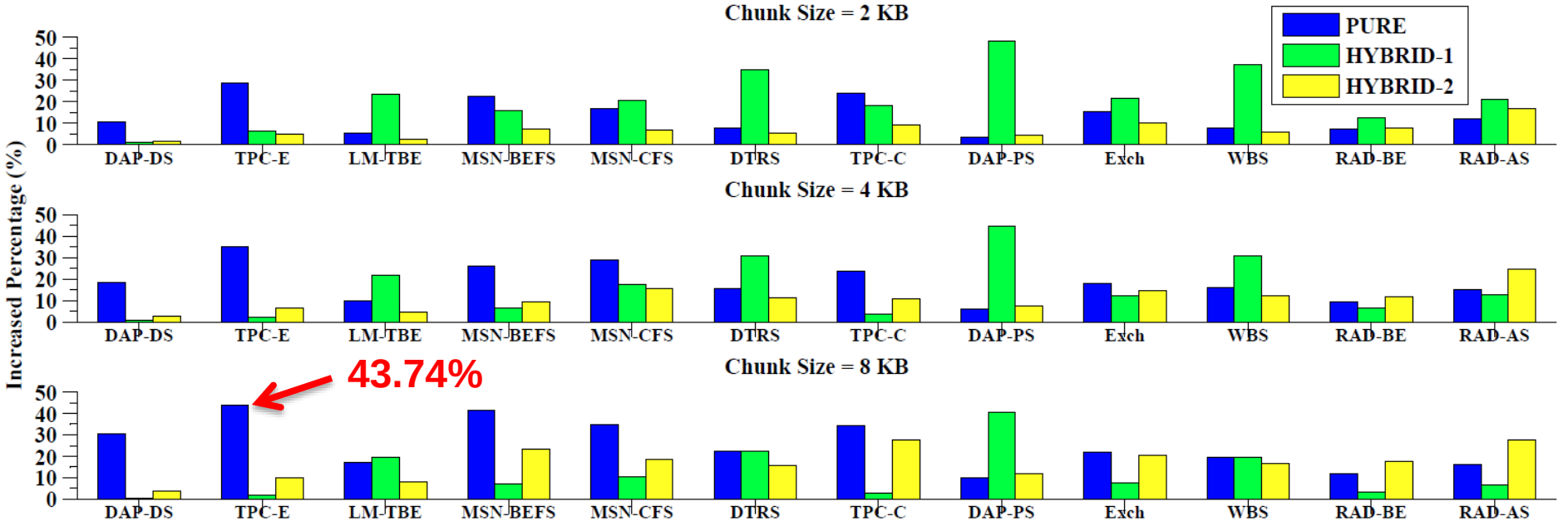
I/O Overhead

- Trace-driven simulation
  - 12 workload traces from production Windows servers
    [Kavalanekar et al., IISWC '08]
  - RAID-6 with n=8 for different chunk sizes

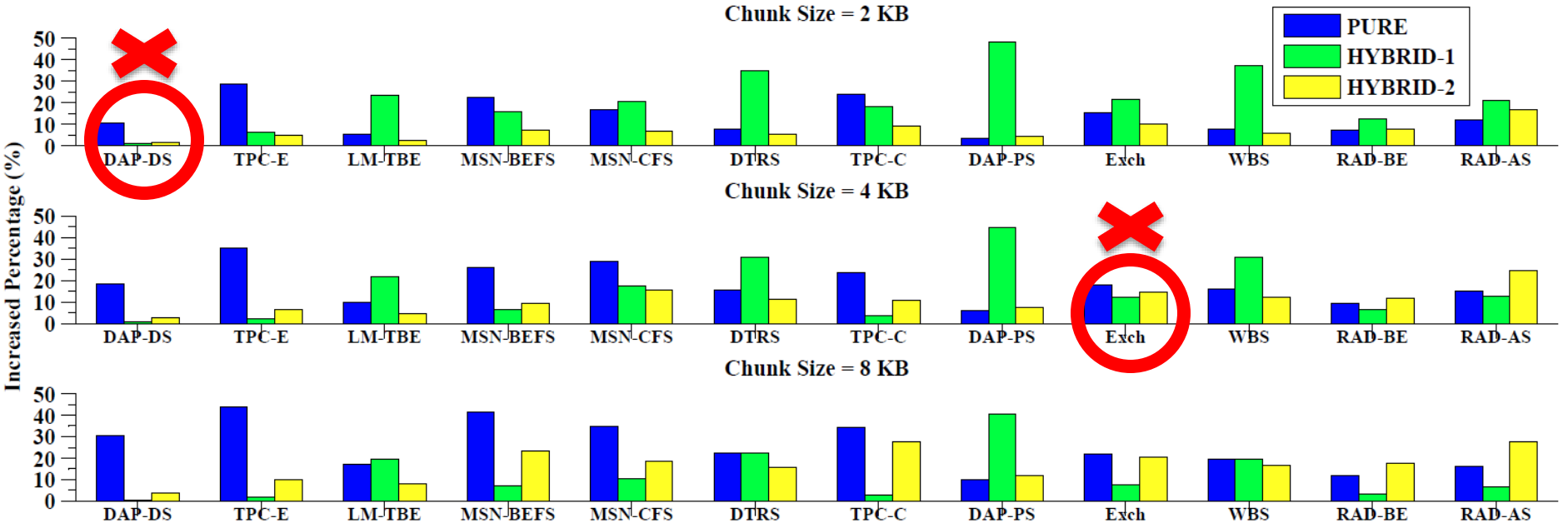


I/O Overhead

- Pure can have high I/O overhead, by up to 43.74%
- I/O overhead can be kept at reasonably low (**often below 15%**) using the best of Hybrid-1 and Hybrid-2, due to I/O gain in subsequent reads
- More discussions in the paper



Choosing the Right Scheme



- Accuracy rate: 34/36 = **94.44%**
- For the two inconsistent cases, the I/O overhead difference between Hybrid-1 and Hybrid-2 is small (**below 3%**)

Implementation Issues

- Implementation in RAID layer:
  - Leverage RAID redundancy to recover from silent data corruptions
- Open issues:
  - How to keep track of first reads and subsequent reads?
  - How to choose between Hybrid-1 and Hybrid-2 based on workload measurements?
  - How to integrate with end-to-end integrity protection?

Conclusions

- A systematic study on I/O-efficient integrity protection schemes against silent data corruptions in RAID systems
- Findings:
  - Integrity protection schemes differ in I/O overheads, depending on the workloads
  - Simpler integrity checking can be used for subsequent reads
- Extensive evaluations on computational and I/O overheads of integrity protection schemes