VIRGINIA TECH

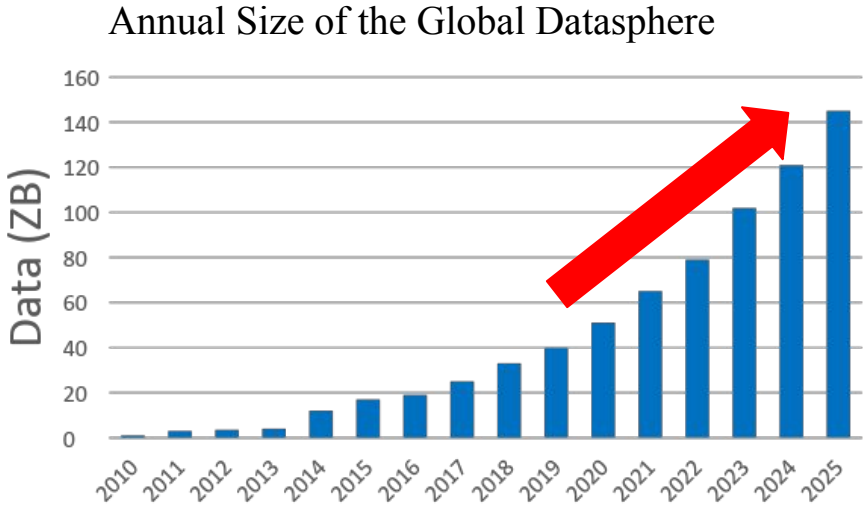
Exploring the Boundaries of the Operating System in the Era of Ultra-fast Storage Technologies

R. Madhava Krishnan

Executive Summary

- Storage systems are **becoming faster, smarter, and memory-centric** to keep up with the rising data generation
- **Operating System (OS)** manages the storage hardware and provides an interface for the applications to access the storage
- Emerging ultra-fast storage technologies are exposing the **performance, programmability, and scalability limitations** in the OS storage stack
- ***This thesis proposes novel kernel-bypass designs to overcome the performance, programmability, and scalability limitations in the OS storage stack***
- This thesis attempts ***to redefine the traditionally demarcated boundary between the OS and the application*** to utilize the emerging ultra-fast storage technologies to its fullest potential

The amount of data we are generating and analyzing is growing exponentially



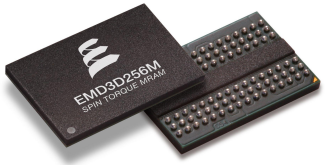
- *IDC expects **175 zettabytes** of data worldwide by 2025*
- *One person created 1.7**MB of data every second** in 2020*
- *We are storing **30% to 40% more data** every year*
- *Data stored is **doubling** every 2 to 3 years*

More data requires more computation and more storage.** The challenge is not limited to storing data but to process and manage it in a **faster and cost effective way.

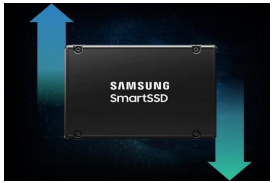
- How Much Data Is Created Every Day in 2021?
- Connected Devices Will Generate 79 Zettabytes of data by 2025

Storage technologies are evolving to keep up with rising data demand

*New **memory-centric** storage technologies are **fast** -- offer **memory like latency** and **storage like persistence***



*Some storage technologies are **compute capable** -- not just store the data but **perform compute** on the stored data **independent of the CPU***

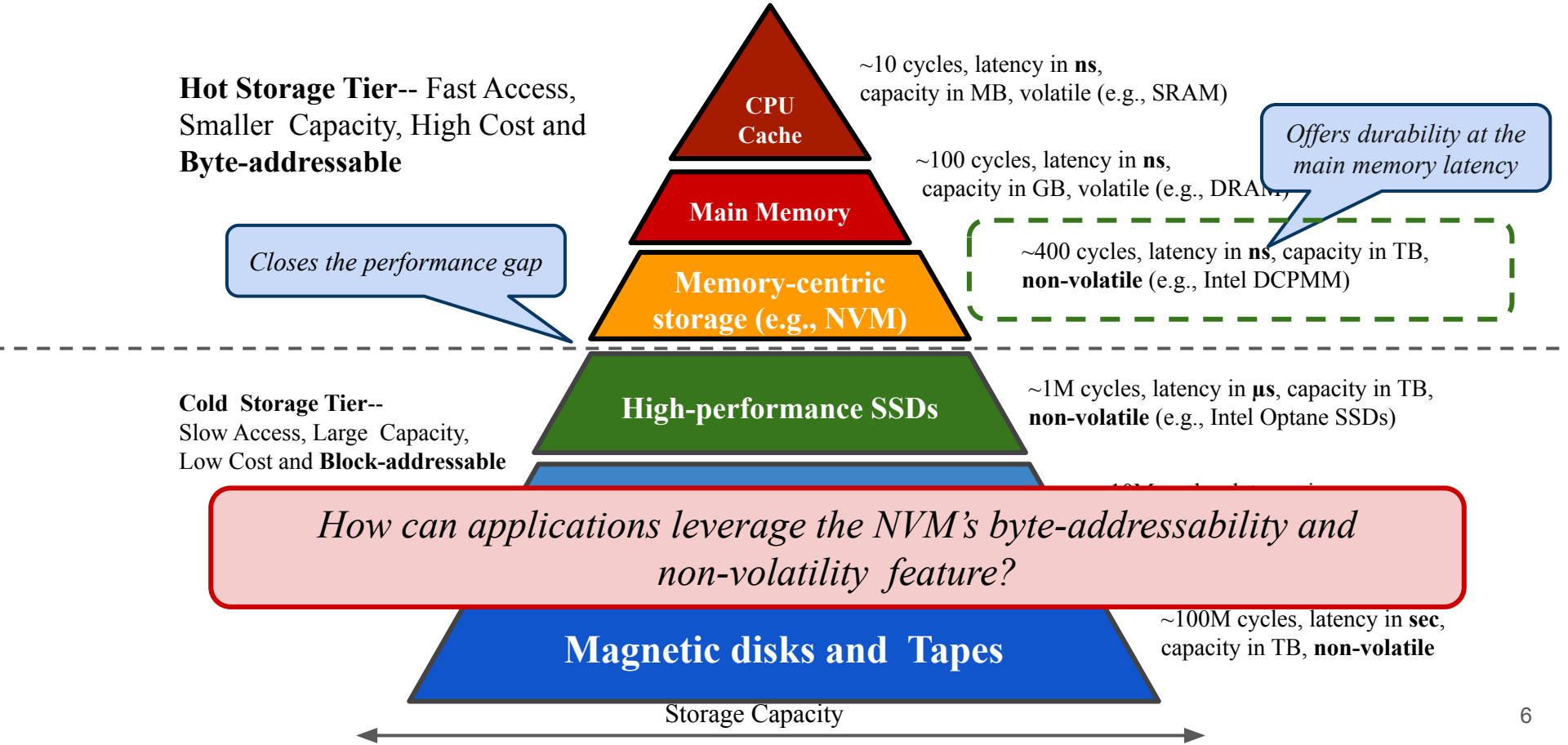


Background on memory-centric storage technologies

- **Byte-addressable** like the DRAM
- **Persistent** like the NAND flash
- Operates at **nanosecond latency**, ~100X faster than NAND flash
- **Lower $/GB** than DRAM, ~2x cheaper

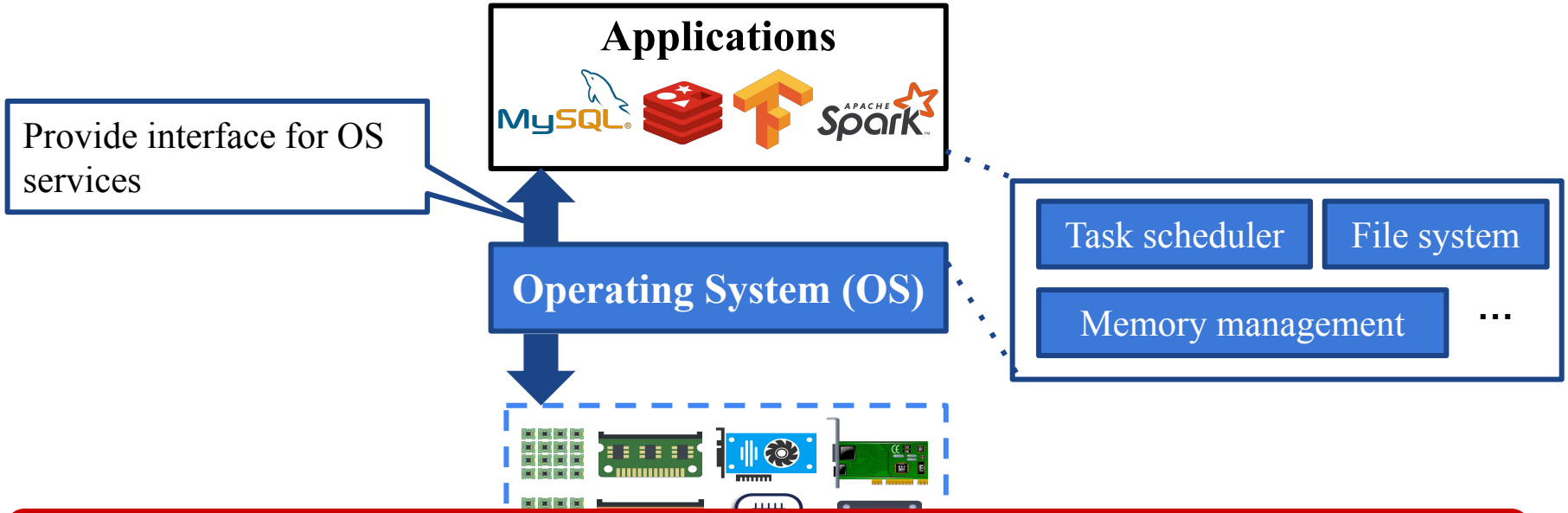
*We consider Intel's **byte-addressable NVM** as a representative memory-centric storage technology*

NVM closes the performance gap between memory and storage



Operating system (OS) manages the storage hardware

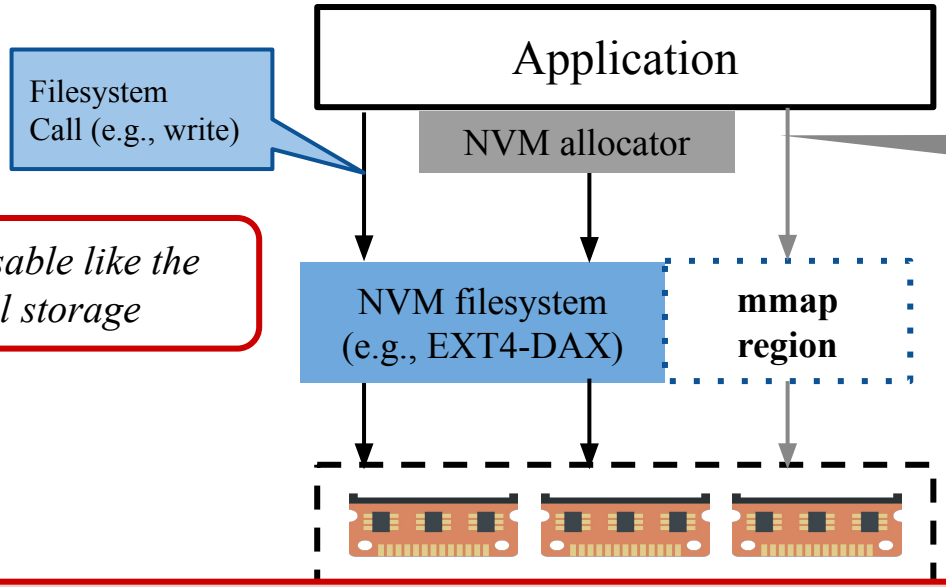
*OS provides an excellent **filesystem based programming abstraction** for the applications to effectively read and write to the storage hardware*



OS storage stack becomes a critical bottleneck for the applications using memory-centric storage

OS storage stack becomes a bottleneck for byte-addressable NVM

***Programmability**: OS storage stack does not support byte-addressable programming*



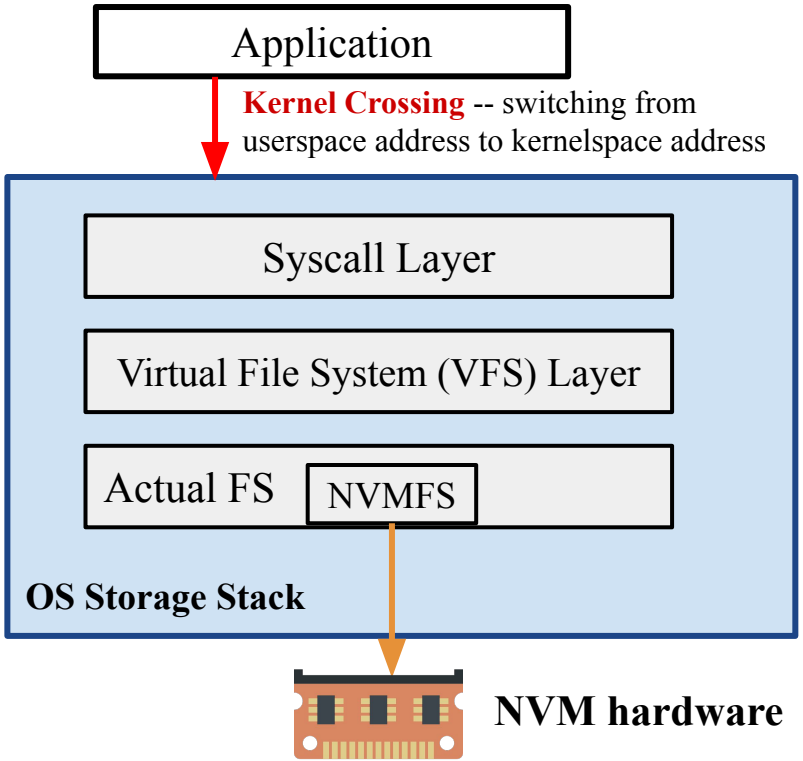
Block addressable like the traditional storage

NVM is mapped to the userspace to enable byte-addressable access

*Applications need to **bypass the OS storage stack** to exploit the byte-addressability feature --* ***this complicates the application programming***

OS storage stack bottleneck -- Latency overhead

***Performance**: OS storage stack increases the NVM access latency*



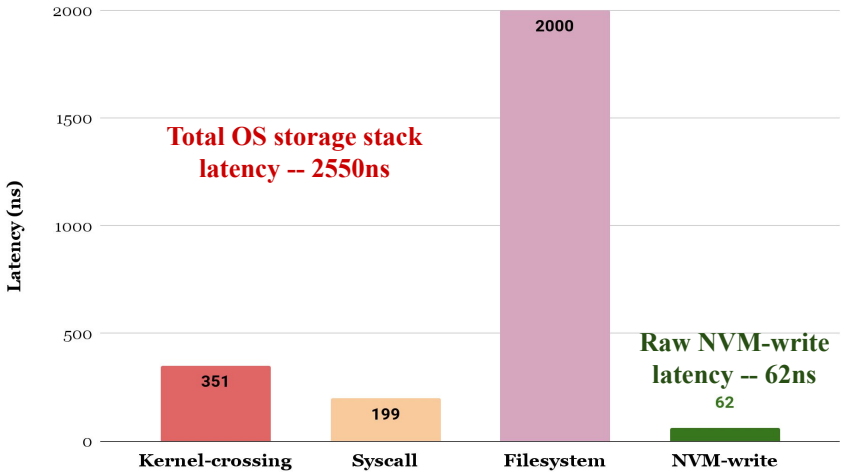
*Applications needs to go through **multiple software layers in the OS storage stack to access NVM** -- Each layer adds additional latency to access NVM*

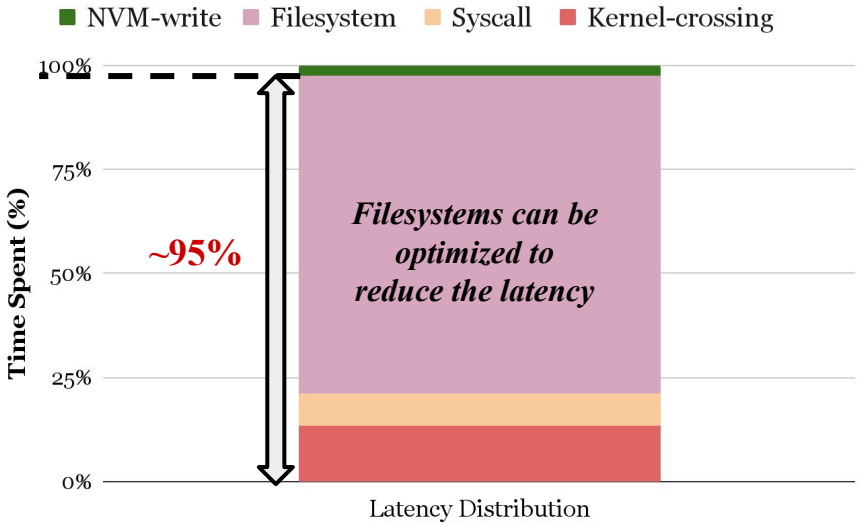
++ latency due to kernel crossing
++ syscall overhead
++ filesystem overhead

Latency Data Source:
An Empirical Guide to the Behavior and Use of Scalable Persistent Memory
XRP: In-Kernel Storage Functions with eBPF

OS storage stack bottleneck -- Context switching overhead

***Performance**: OS storage stack increases the NVM access latency*



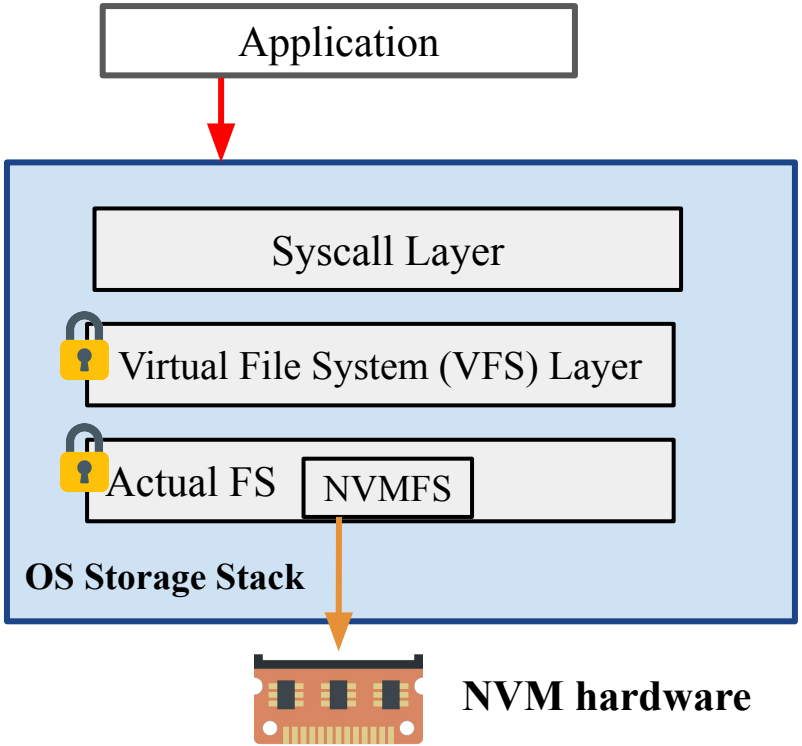


*OS storage stack latency is **significantly greater** than the NVM media's write latency -- **NVM write latency (62ns) is increased by 9X even with zero filesystem overhead (550ns)***

An Empirical Guide to the Behavior and Use of Scalable Persistent Memory
XRP: In-Kernel Storage Functions with eBPF

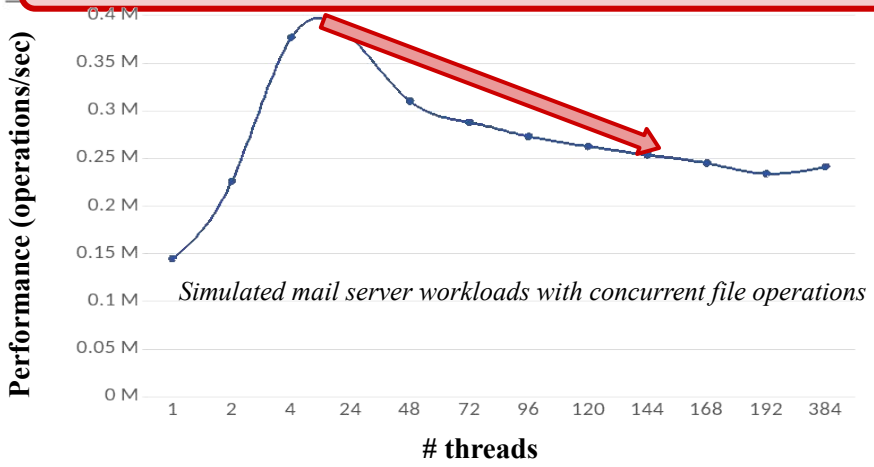
OS storage stack bottleneck -- Scalability bottleneck

***Scalability**: OS storage stack becomes a scalability bottleneck for applications*



FS	Bottleneck	Sync. object	Scope	Operation
VFS	Rename lock	rename_lock	System	rename()
	inode list lock	inode_sb_list_lock	System	File creation and deletion
	Directory access lock	inode->i_mutex	Directory	All directory operations
	Page reference counter	page->_count	Page	Page cache access
	dentry lockref [38]	dentry->d_lockref	dentry	Path name resolution
	Acquiring a write lock for a B-tree node	btrfs_tree_lock()	File system	All write operations

Poorly scalable lock based concurrency control



Simulated mail server workloads with concurrent file operations

Understanding manycore scalability of filesystems [ATC-16]

Thesis Overview: Challenges, vision, contributions, and outcomes

Thesis Question and solution

How to explore the memory-centrict storage technologies like the byte-addressable NVM without constrained by the performance, programmability, and scalability limitations of the OS storage stack?

Kernel-Bypass storage stack: *Bypass the OS storage stack and implement the* ***storage stack in the userspace***

Kernel-bypass storage stack

++ No context switching overhead

++ No syscall overhead

++ **Potential to** enable byte-addressability by **deviating from filesystem abstraction**

Kernel-bypass storage stack forgoes isolation

Kernel-bypass storage stack

++ No context switching overhead

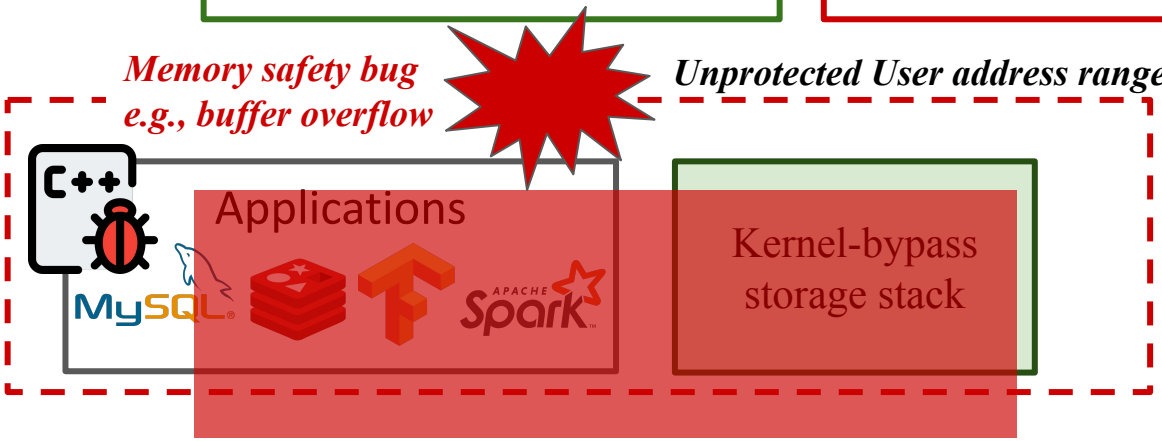
++ No syscall and filesystem overhead

++ Enables byte-addressability

-- Forgoes kernel-userspace isolation

-- Memory safety vulnerabilities

-- Needs an efficient programming model



Buggy application can corrupt the kernel-bypass storage stack

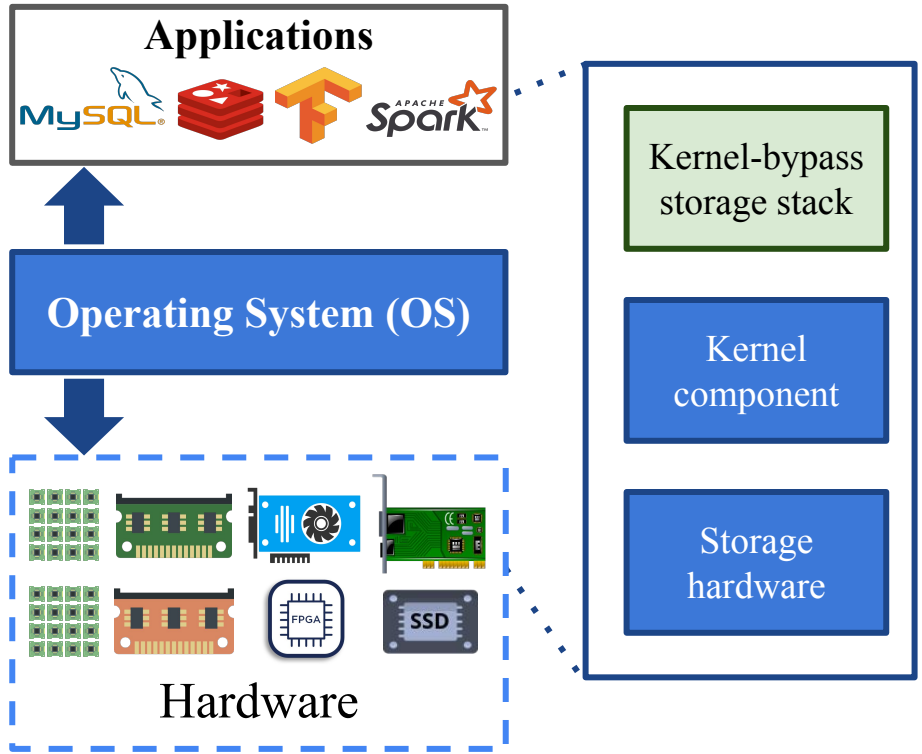
OS enforces isolation from buggy applications

Challenges in designing kernel-bypass storage stack

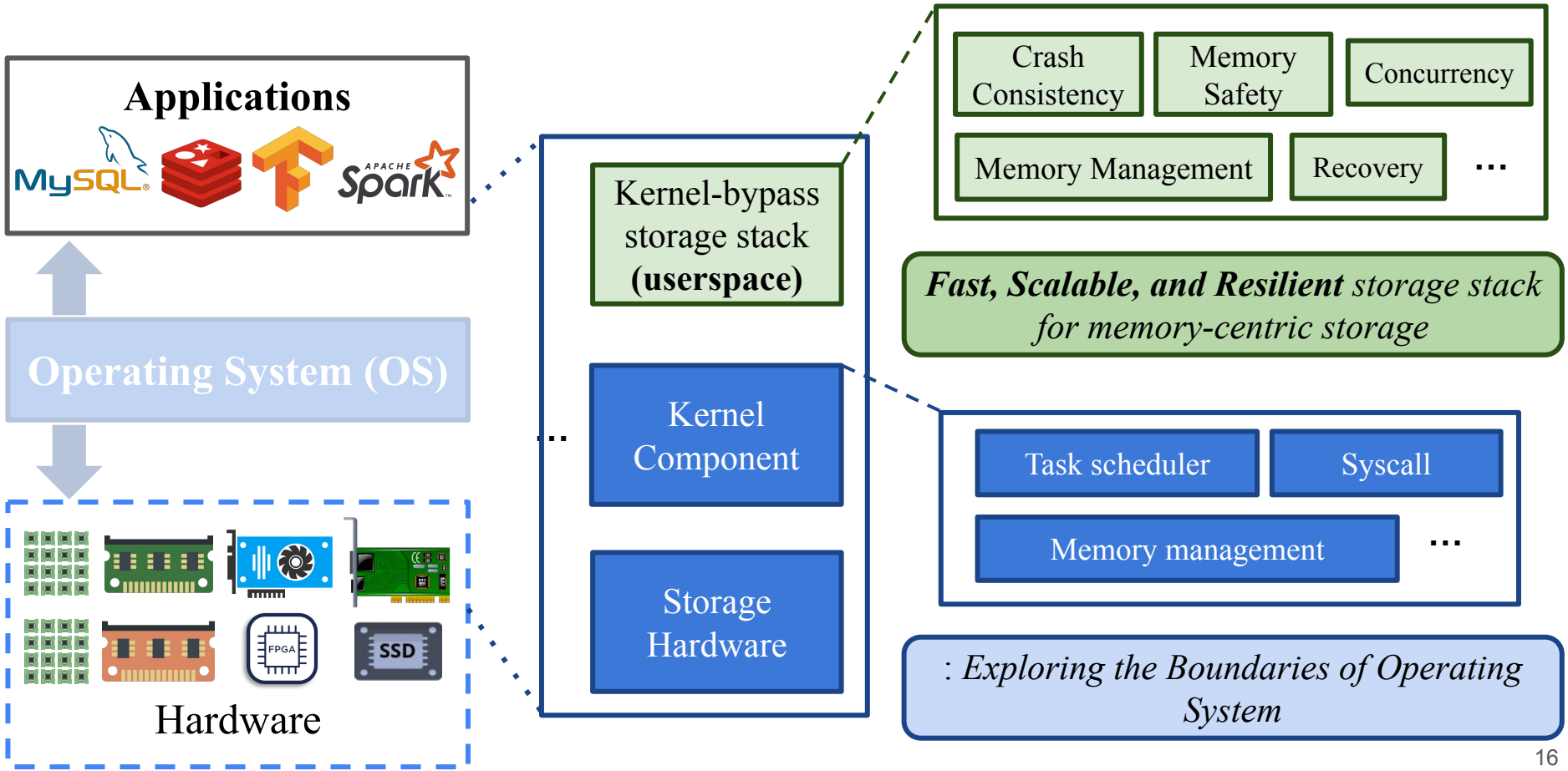
Programmability: byte-addressable and crash consistent programming model

Scalability: Efficient concurrency model to scale performance with increasing core count

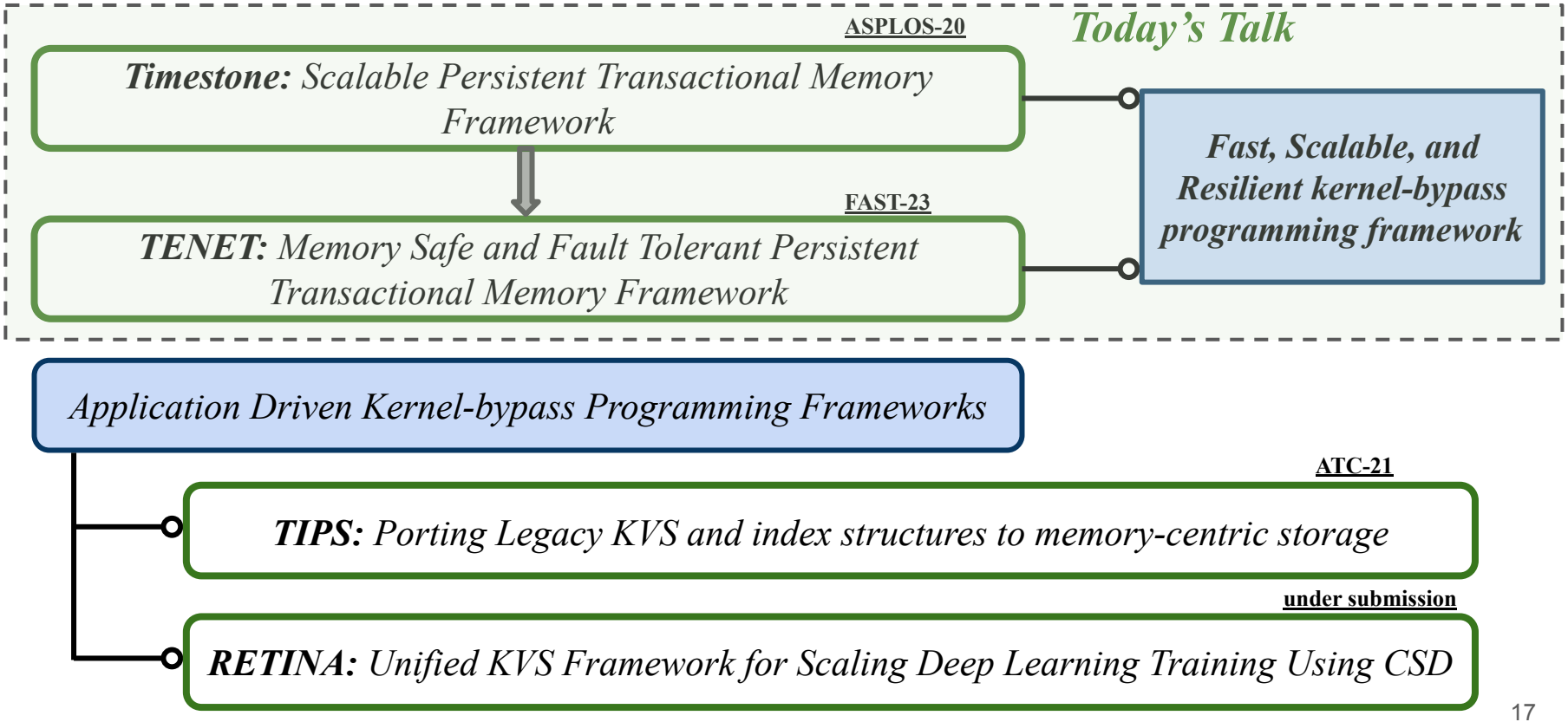
Resilience: Enforce protection against software bugs and hardware errors



Thesis vision: Kernel-bypass storage stack architecture



Thesis contributions: Kernel-bypass techniques for memory-centric storage



Thesis outcome: Feature comparison with OS storage stack

Feature	OS storage stack	Kernel-bypass storage stack [TENET]
Programming abstraction	**Filesystem**	Persistent Transaction
Programming support	**No, only block-adderssable**	Yes, byte and block-addressable
Multi-core scalability	**Poorly scalable**	Scales across 100's CPU cores
Latency scale	**Micro-second scale**	Nano-second scale
Isolation and Safety	**Yes**	Yes, also fault tolerant

Why transactional programming abstraction?

How to make transactional programming scalable and high-performant?

How to make transactional programming memory safe and fault tolerant?

Durable Transactional Memory Can Scale With Timestone

In the Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems ***(ASPLOS 2020)***

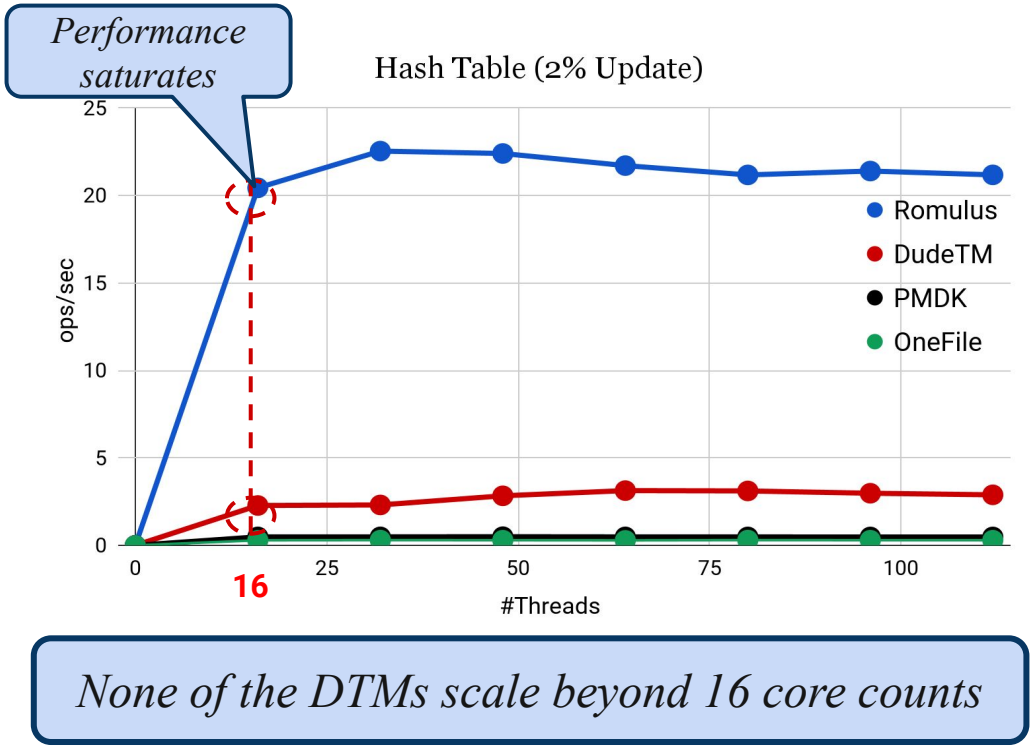
Transactional programming model for byte-addressable NVM

Why transactional programming model?

- Transactions support ***byte-addressable access***
- ***Concurrent programming is easy*** with transactional model
- Software Transactional Memory (STM) designed for concurrent DRAM programming
- ***Persistent Transactional Memory (PTM)*** extends STM to support crash consistent updates to storage
- ***Applications using PTM do not need to worry about concurrency and crash consistency***

Applications have to define a serial ***single-threaded code region*** *and the transaction library guarantees* ***concurrent and crash consistent execution***

Prior PTMs suffer from high overhead and poor scalability



None of the DTMs scale beyond 16 core counts

Poorly scalable concurrency control

DudeTM → tinySTM, **Global locking table**

PMDK → **Readers-writer lock**

Romulus → **single writer combining**

High Write Amplification (WA) due to crash consistency operations

DudeTM → **REDO logging, 4X WA**

PMDK → **REDO and UNDO logging, 70X WA**

Romulus → **Data mirror, 2X WA**

DudeTM: Building Durable Transactions with Decoupling for Persistent Memory [ASPLOS -17]
Romulus: Efficient Algorithms for Persistent Transactional Memory [SPAA-18]
PMDK: Persistent Memory Development Kit [Intel]
OneFile: Onefile: A wait-free persistent transactional memory [DSN-19]

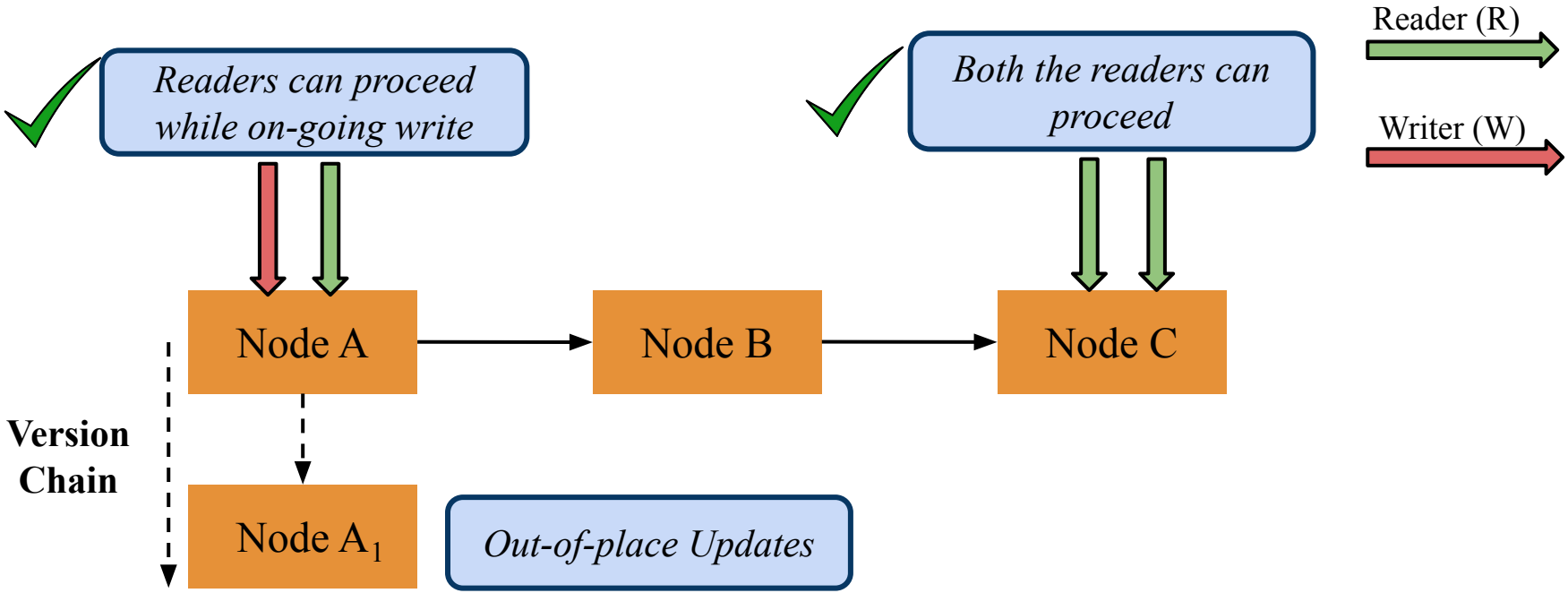
Transactional programming model can scale with Timestone

*How to make transactional programming **scalable and high-performant?***

***Timestone:** Multi-version Concurrency Control (MVCC) based Persistent Transactional Memory (PTM)*

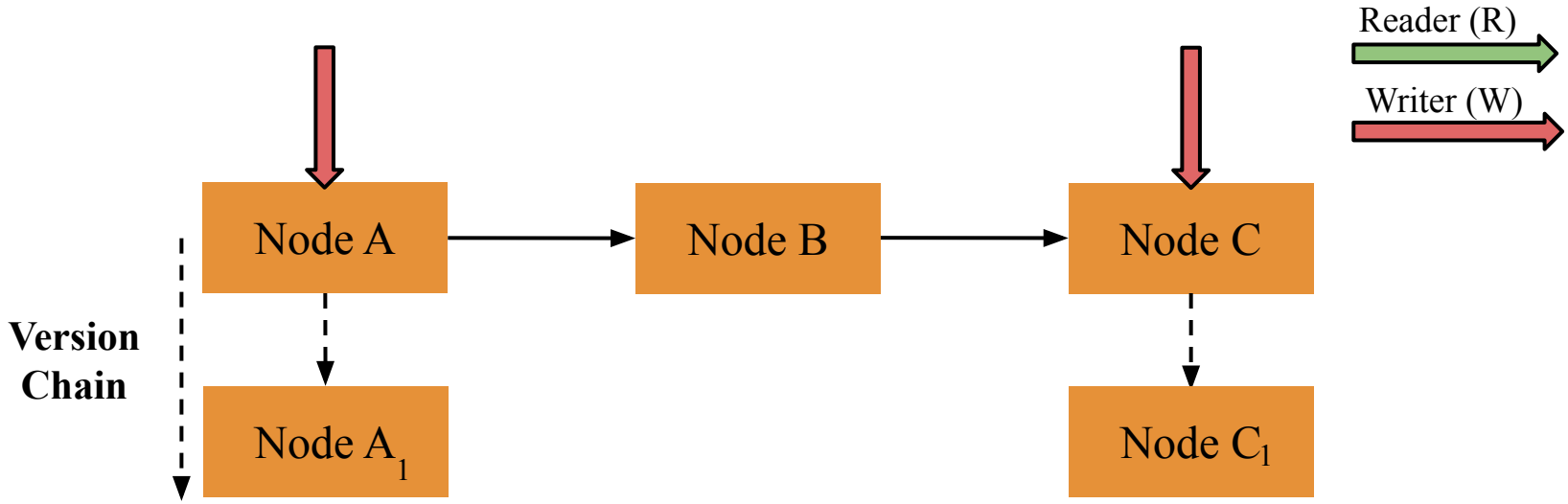
- *Adopt **MVCC for multi-core scalability** by supporting non-blocking reads and disjoint writes*
- ***TOC Logging for** high-performance and efficient crash consistency by **reducing write amplification***

MVCC supports concurrent non-blocking reads



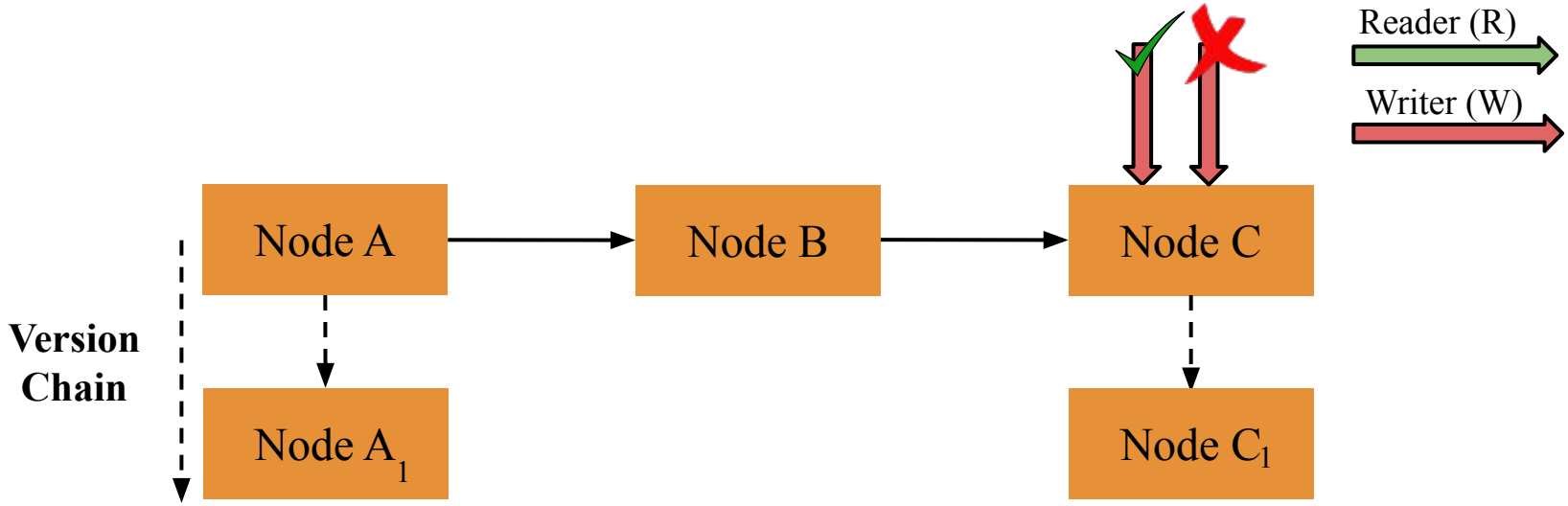
Readers** are always guaranteed to **see a consistent version** of an object and hence there is **no read-write conflict

MVCC supports concurrent disjoint writes



Non-conflicting writers can perform concurrent execution

MVCC supports concurrent disjoint writes



conflicting writers are serialized for correctness

TOC Logging for efficient crash consistency

*Traditional journaling techniques like **UNDO and REDO logging causes high write amplification***

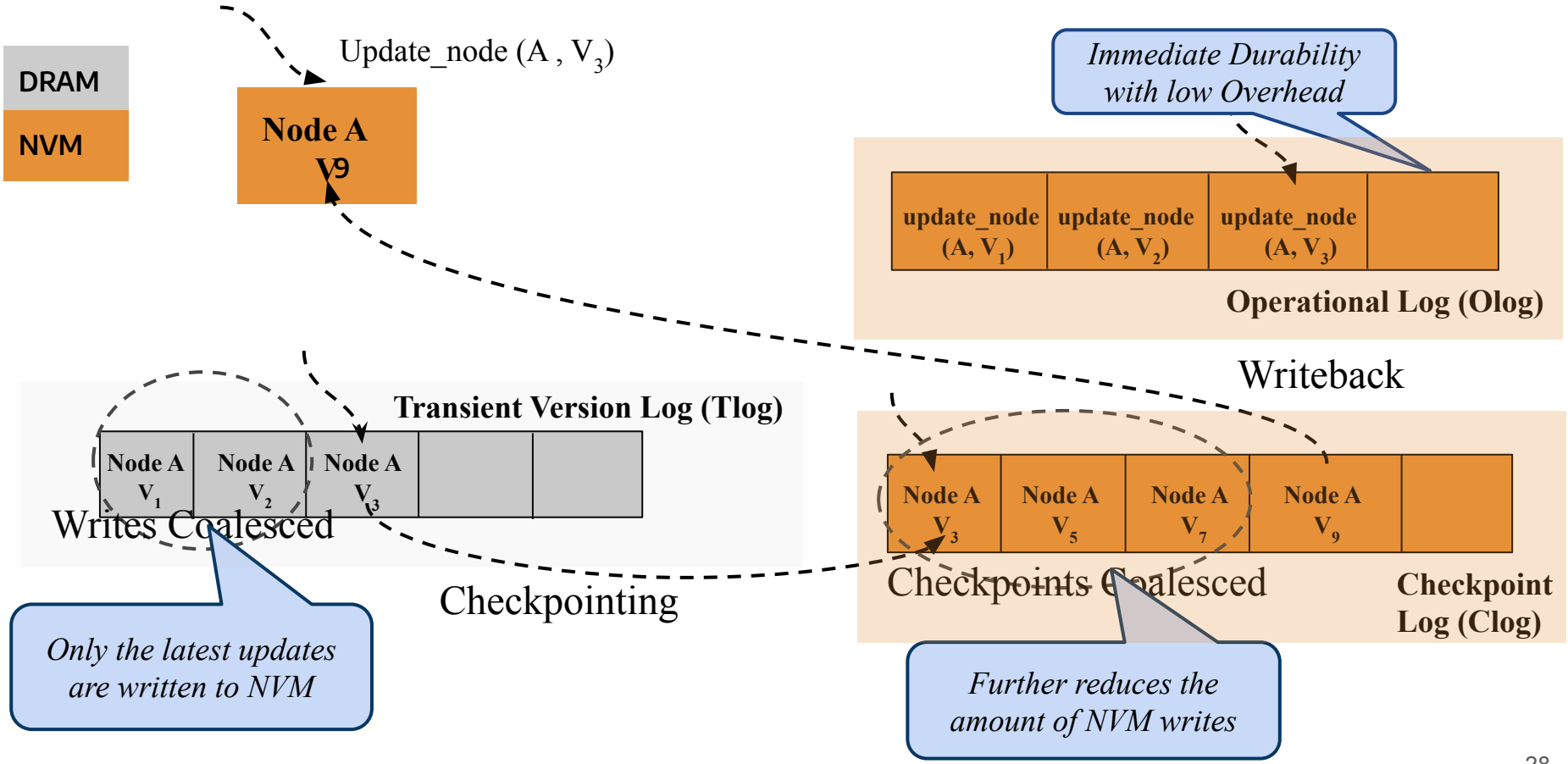
***MVCC also causes high write amplification** as it creates new version on every write operation*

TOC Logging:** multi-layered hybrid NVM-DRAM logging technique **to reduce write amplification due to logging and MVCC

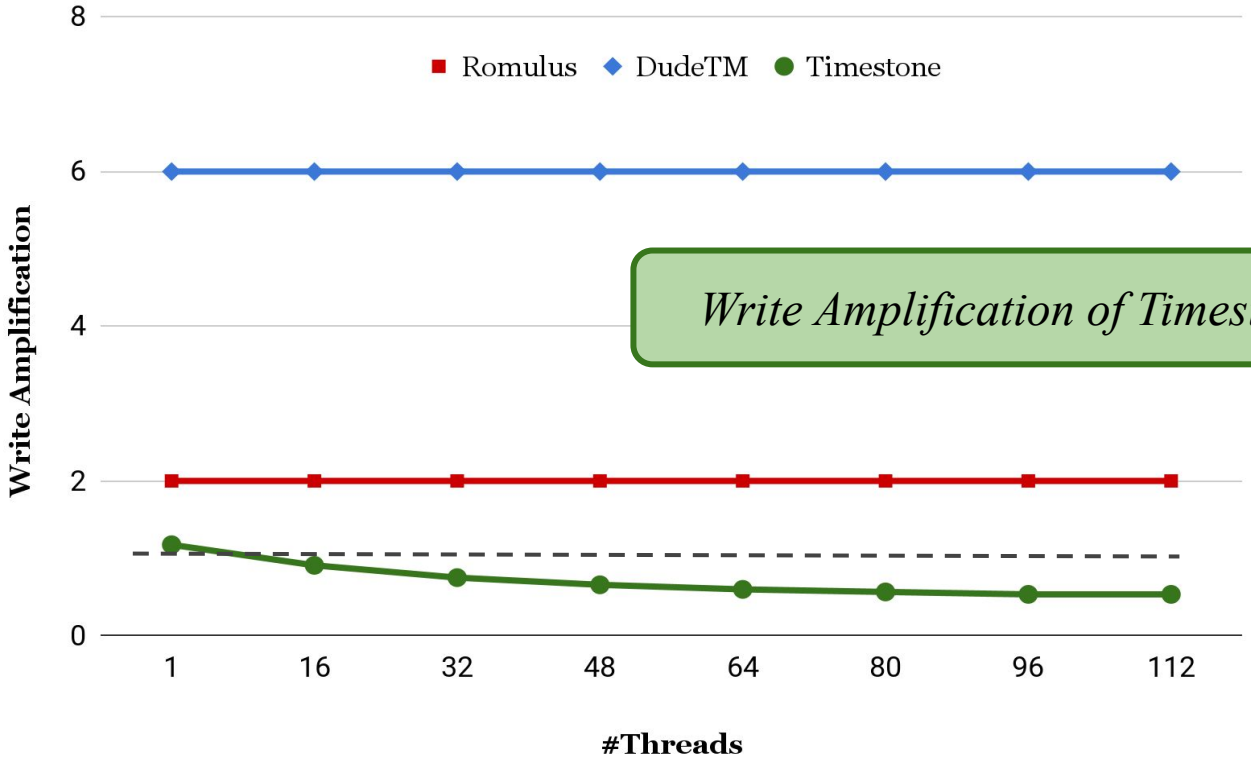
TOC Logging hybrid NVM-DRAM architecture

- **TOC** logging is a multilayered hybrid DRAM-NVM logging
- *Transient Version log on **DRAM** (TLog)*
  - New version during write operation is created on the TLog
  - To leverage faster DRAM and ***enable batching writes to NVM***
- *Operational log on **NVM** (OLog)*
  - Log only the ***operation semantics ("what app did")*** rather than logging the entire data
  - To Guarantee ***immediate durability in the critical path***
- *Checkpoint log on **NVM** (CLog)*
  - Periodical ***snapshot of TLog is checkpointed*** on the CLog
  - To guarantee correct recovery and fast recovery time

TOC Logging in Action

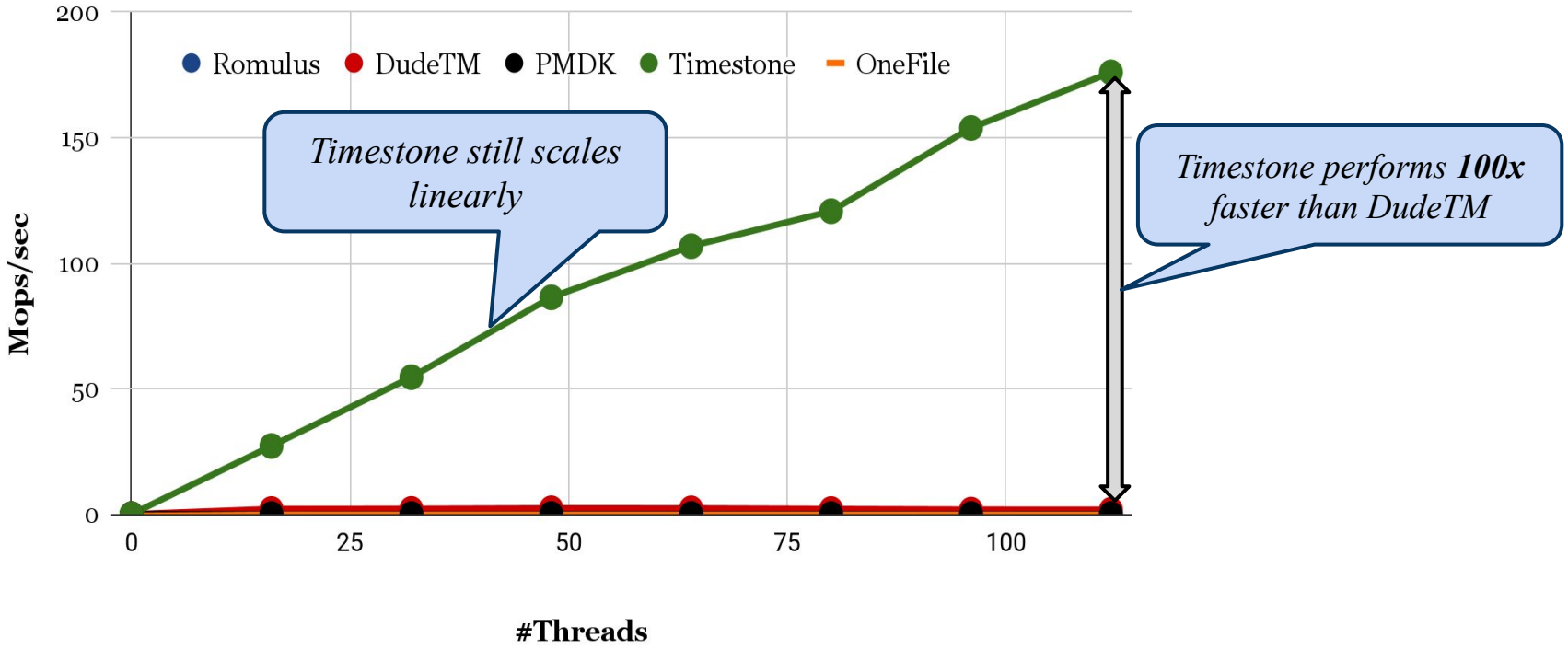


Write amplification for hash table (80% Update)

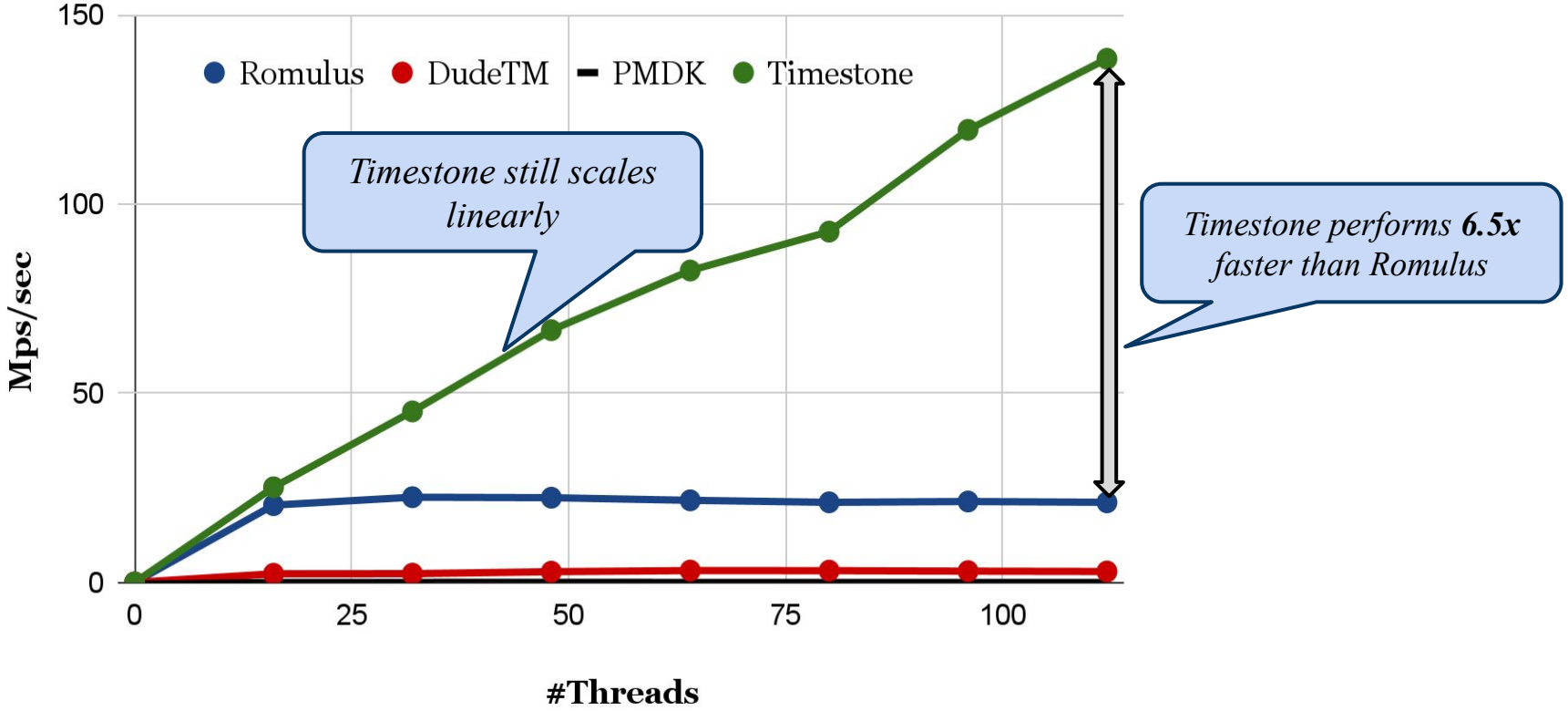


Write Amplification of Timestone is always <= 1

Scalability for write-intensive hash table (80% Update)



Scalability for read-intensive hash table (2% Update)



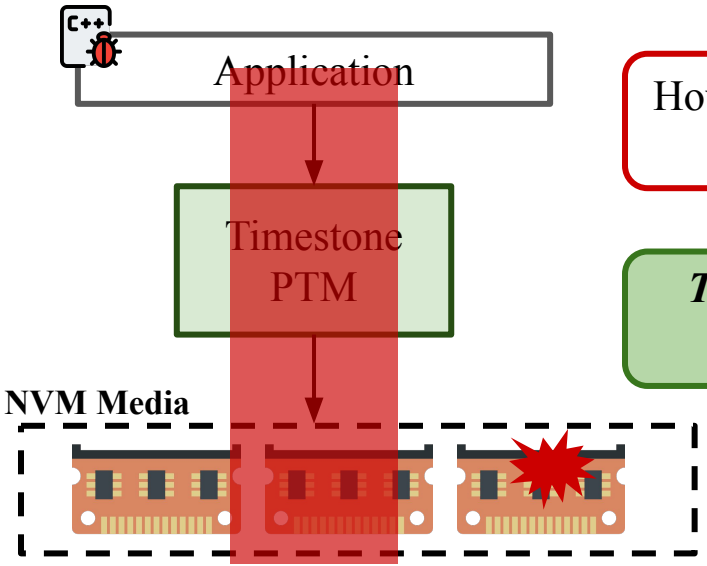
Summary of Timestone's features

Timestone PTM

++ *Scalable for 100's of CPU cores*

++ *High performance for different data structures and workloads*

-- *Vulnerable to software bugs and media errors*



How do we protect the Timestone PTM and the NVM data from **software bugs and media errors?**

TENET: *guarantees* ***memory safety*** *from software bugs and* ***fault tolerance*** *against media errors*

NVM media error

TENET: Memory Safe and Fault Tolerant PTM

In the Proceedings of the 21st Conference on File Systems and Storage Technologies
(FAST 2023)

Memory safety bugs are common and a critical issue!

Home / Innovation / Security

Microsoft: 70 percent of all security bug are memory safety issues

Percentage of memory safety issues has been hovering at 70 percent for the past 12 years.

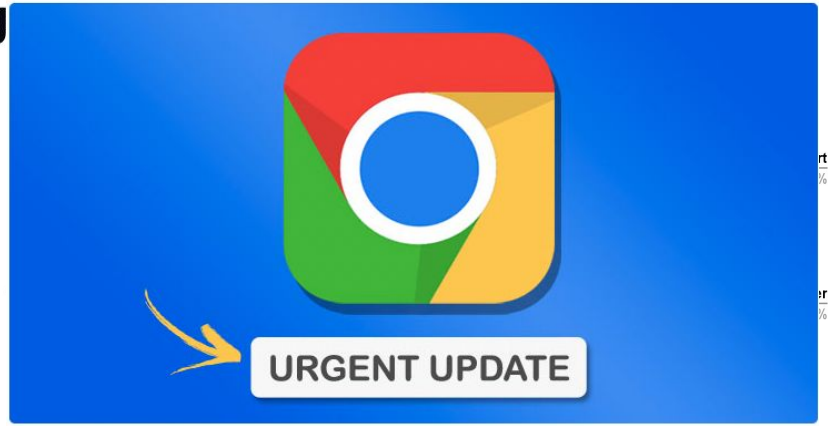
Home / Innovation / Security

Chrome: 70% of all security bugs are memory safety issues

Google software engineers are looking into ways of eliminating memory management-related bugs from Chrome.

Google Chrome Hit by Second Zero-Day Attack - Urgei

Apr 19, 2023 Ravie Lakshmanan Browser Security / Zero-Day



Google on Tuesday rolled out emergency fixes to address another actively exploited high-severity zero-day flaw in its Chrome web browser.

The flaw, tracked as **CVE-2023-2136**, is described as a case of integer overflow in Skia, an open source 2D graphics library. Clément Lecigne of Google's Threat Analysis Group (TAG) has been credited with discovering and reporting the flaw on April 12, 2023.

"Integer overflow in Skia in Google Chrome prior to 112.0.5615.137 allowed a remote attacker who had compromised the renderer process to potentially perform a sandbox escape via a crafted HTML page," according to the NIST's National Vulnerability Database (NVD).

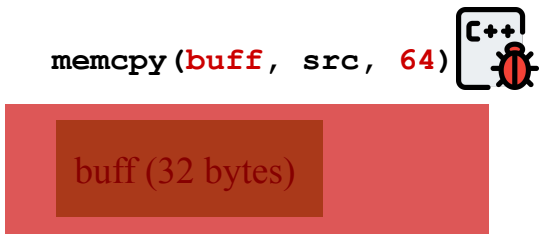
https://www.chromium.org/Home/chromium-security/memory-safety/
https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Background on types of memory safety violations

01 **Memory Safety Violations**

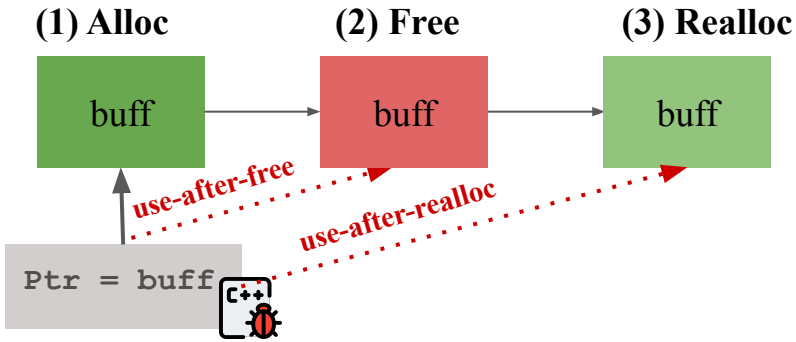
- Spatial Safety Violations
- Temporal Safety Violations

Spatial Safety Violations



*Spatial safety violations happens when applications access the memory **beyond the allocated range***

Temporal Safety Violations



*Temporal safety violations happens when applications access the memory **using dangling pointers***

Prior memory safety works suffer from high performance and cost overhead

PTM	Baseline PTM*	Spatial Safety	Temporal Safety	Fault Tolerance	Performance Overhead	NVM Cost Overhead
Libpmemobj-**R**	libpmemobj	✗	✗	✓	100%	High
SafePM [Eurosys-22]	libpmemobj	✓	✓	✗	55%	
Pangolin [ATC-19]	libpmemobj	✓	✗	✓	67%	Moderate

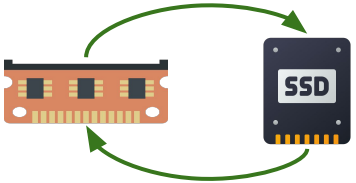
Guaranteeing memory safety and fault tolerance ***at a lower performance overhead and cost is a very challenging problem***

*Performance overhead reported for hash table
*Pangolin's overhead is directly referenced from the paper

TENET overview: Goals and Assumptions

TENET is an NVM programming framework to develop memory safe and fault tolerant NVM data structures and applications

- Protect NVM data from a buggy application code
  - **Guarantee spatial safety and temporal safety**
- Protect NVM data against Uncorrectable Media Errors (UME)
  - **Guarantee a performance and cost efficient fault tolerance**



TENET adopts Timestone PTM as its programming model and adds memory safety and fault tolerance support

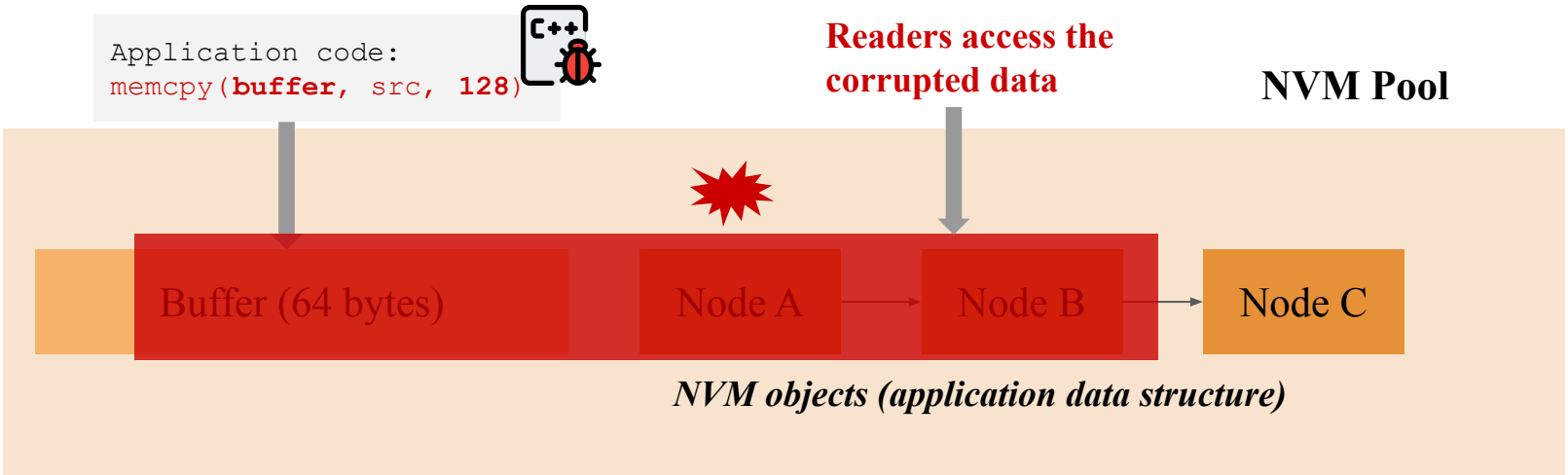
Spatial safety design in TENET

*Application code or any code outside the TENET library is **not allowed** to perform direct NVM writes*

*Only the TENET library code **is allowed** to perform writes to the NVM data*

Direct NVM writes in the application code is dangerous

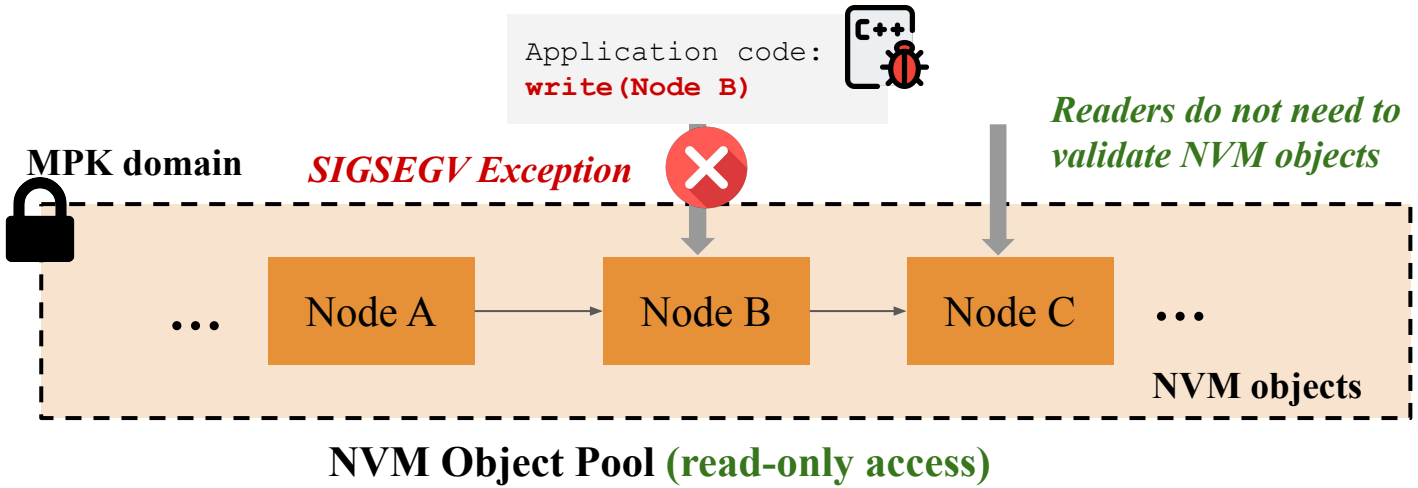
A buggy application write on the NVM can cause spatial safety violation



***NVM is read-only** for the application to **prevent buggy writes** from corrupting the NVM data*

Prevent direct NVM writes using Memory Protection Keys (MPK)

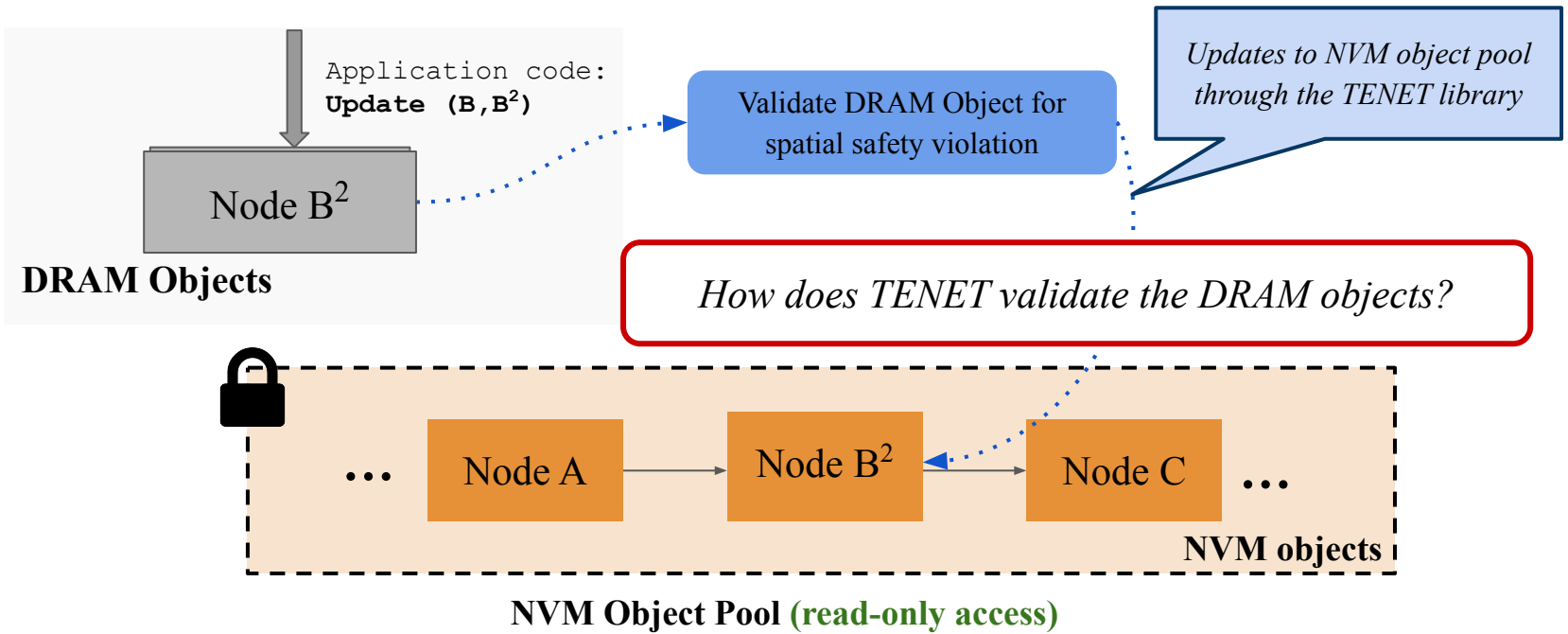
*TENET uses **MPK to enforce read-only access** to the NVM object pool for all the code outside of the TENET library*



How does application write to the NVM objects?

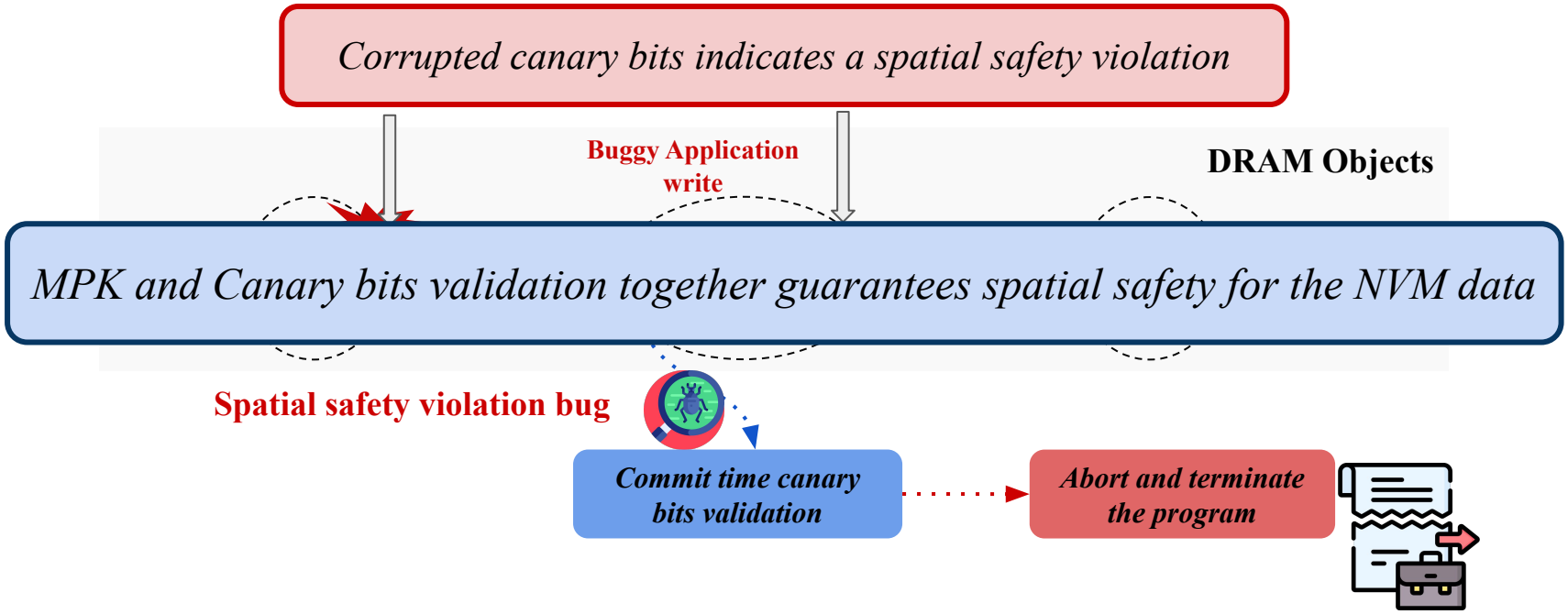
Prevent direct NVM writes using Memory Protection Keys (MPK)

Applications write only on the DRAM region *and* ***TENET writes back*** *the DRAM object to the NVM* ***after validating it for spatial safety***



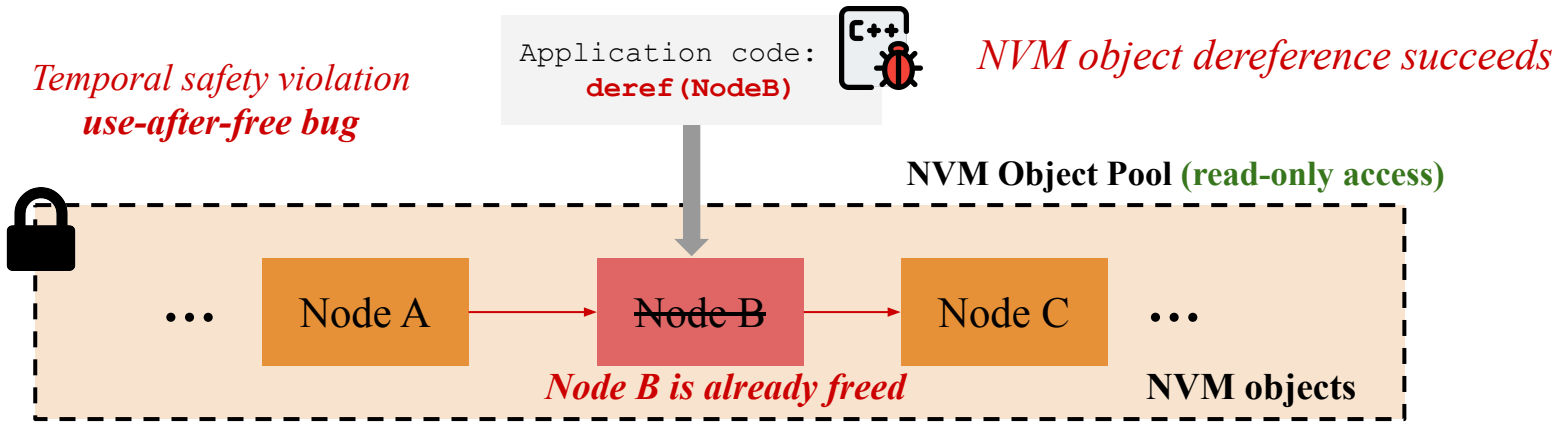
Protecting DRAM objects using canary bits

*TENET assigns **8 byte canaries** at the boundary of a DRAM object and the canary bits are inspected when the application **commits the transaction***



Read-only NVM access can cause temporal safety violations

Doe ***Applications can dereference a dangling pointer*** *to an NVM object as TENET grants read access to the NVM* *ption?*

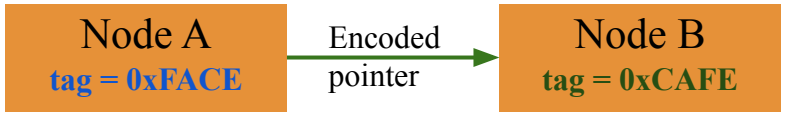


How does TENET enforce temporal memory safety for the NVM objects?

Enforcing temporal safety for NVM objects using pointer tags

*NVM address is **tagged at the time of creation**; the tag is stored in the allocated NVM object and a copy of the **tag is encoded in the upper 16 bits of the NVM pointer***

The encoded pointer to Node B is stored in Node A



tag is stored in the NVM object at the time of creation

Upper 16 bits are unused

- **Node B's address** → 0x00001265FFCAB734; **Tag** → 0xCAFE
- **Encoded pointer** → Node B || Tag << 48 → **0xCAFE1265FFCAB734**

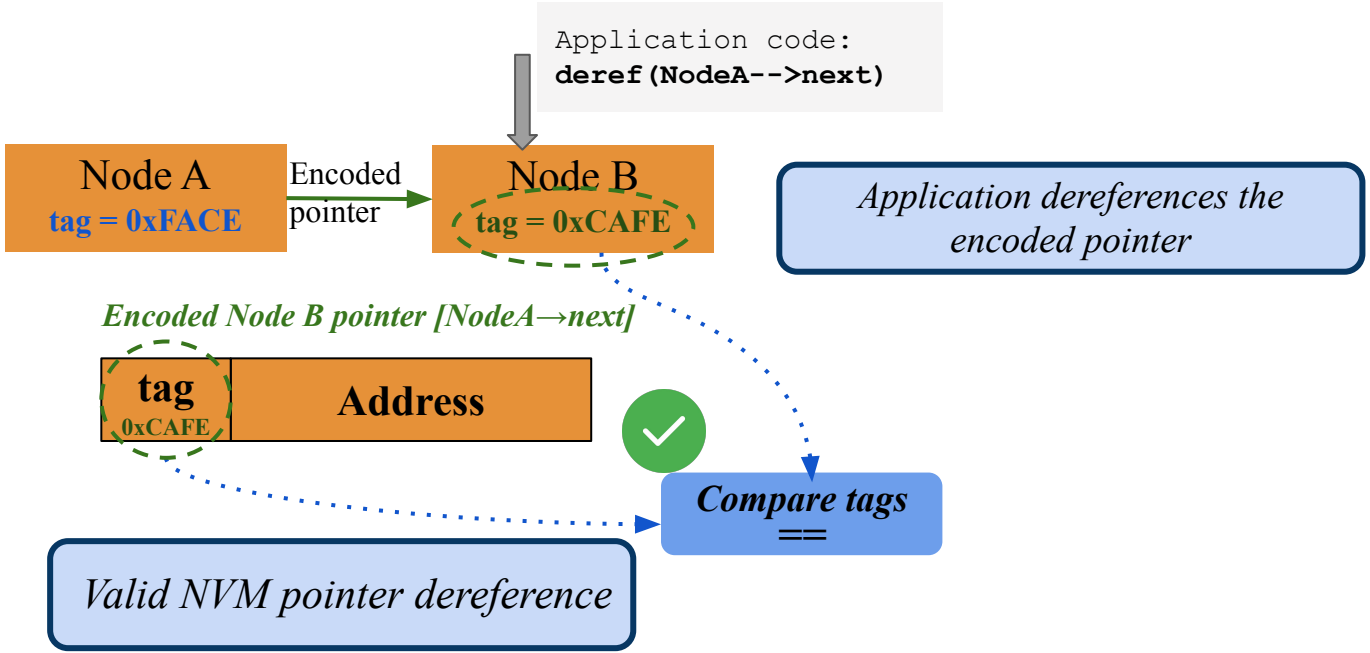
tag bits **NVM address**

A copy of the tag is encoded to the upper 16 bits of Node B's address

Encoded pointer layout

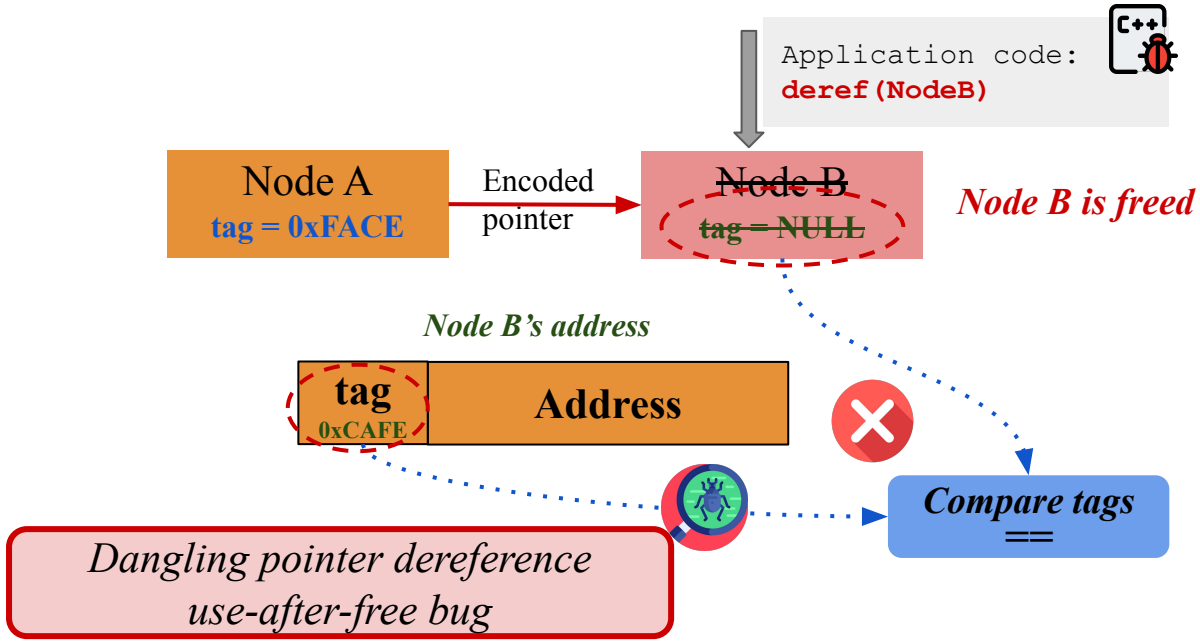
Enforcing temporal safety for NVM objects using pointer tags

***Application accesses the NVM objects using the encoded pointer** -- the encoded tag in the pointer is compared with the tag stored in the corresponding NVM object*



Enforcing temporal safety for NVM objects using pointer tags

Dangling pointer is detected *by comparing the tag stored in the NVM object with the tag encoded in the pointer to the NVM object*



Memory safety techniques are performance efficient

*MPK and pointer tags **do not incur** any additional crash consistency overhead*

- *__MPK__ is a userspace __hardware primitive__ -- __< 20 cycles__ to switch permissions* — ++ **Performance**
- ***Pointer tags** are embedded __in the address__ -- no __centralized metadata__ management* — ++ **Scalability**

*Every NVM load and store **do not need to be checked** for memory safety violations*

- ***Spatial safety** checks of DRAM objects only at the __commit time__ -- new objects are __not visible__ before transaction commits* — ++ **Performance**
- ***Temporal safety** checks of NVM objects only at __first dereference__ -- a derefrenced object __cannot be freed__ by a concurrent write transaction* — ++ **performance**

Replicating NVM data for fault tolerance against UME

- **NVM data corruption due to software errors**
  - Spatial memory safety → MPK + canary bits validation ✓
  - Temporal memory safety → Pointer tags validation ✓

How does TENET make the NVM data fault tolerance against the UME?

- TENET replicates the NVM data to the local SSD to maintain backup copy
- **Restore the corrupted NVM page from the SSD replica**
- TENET's replication provides many desirable properties
  - **Cost efficiency** → replicating to the local SSD
  - **Performance efficiency** → replicating the data out-of-the critical path
  - **Consistent loss-less recovery**

Evaluation of TENET

How does TENET compare against the prior PTM works in terms of features and performance overhead?

Evaluation Settings

- We use a 2 socket server with 64 core Intel Xeon Gold CPU
  - 64GB DRAM, 512GB NVM, 1TB SSD
- We evaluate two different versions of TENET
  - **TENET-MS → supports only memory safety**
  - **TENET → supports memory safety and fault tolerance**
- We evaluate TENET with different data structures for different read/write ratios
  - YCSB workloads and microbenchmarks

Comparison of TENET with the other PTMs

PTM	Baseline PTM*	Spatial safety	Temporal Safety	Fault tolerance
Libpmemobj-R	libpmemobj	✗	✗	✓
SafePM [Eurosys-22]	libpmemobj	✓	✓	✗
Pangolin [ATC-19]	libpmemobj	✓	✗	✓
TENET	Timestone	✓	✓	✓

*Replicates NVM data to a local NVM pool -- **High cost overhead***

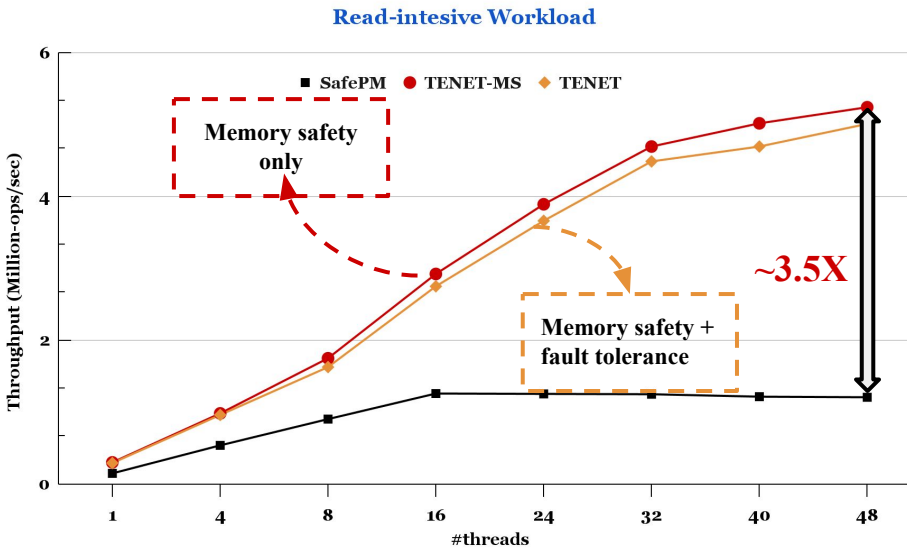
Uses address sanitizer (Asan) -- ***High performance overhead***

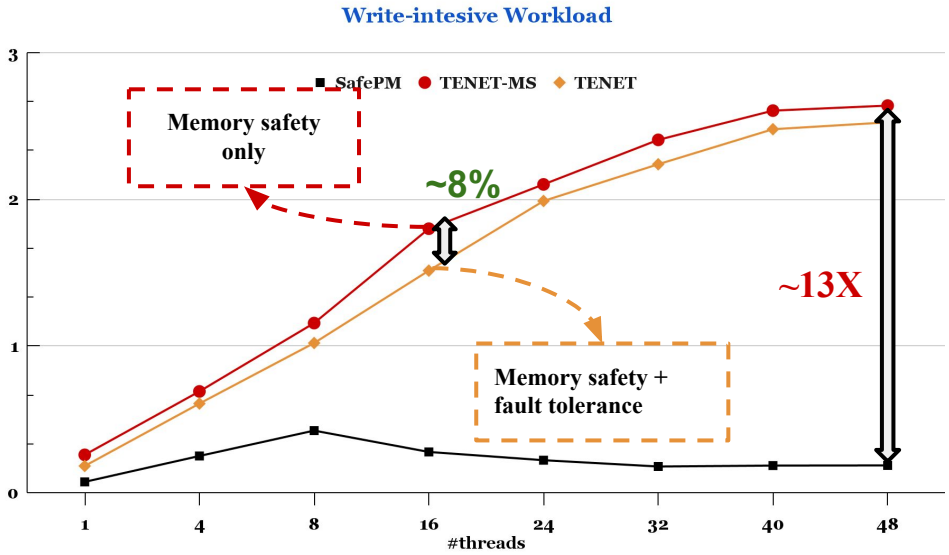
Supports parity based replication and object checksums -- ***High performance overhead due to increased write amplification***

*TENET is the only PTM to provide **spatial memory safety, temporal memory safety, and fault tolerance** for the NVM data*

*PTM - persistent transactional memory
*Libpmemobj is a transactional library in the PMDK

Performance of TENET and SafePM for hash table





*TENET is up to **13x faster** than SafePM*

*Adding replication **incurs only a 8% performance overhead***

TENET does not compromise the scalability of the Timestone PTM

Other Thesis Contributions

TIPS: Framework for porting legacy KVS to NVM

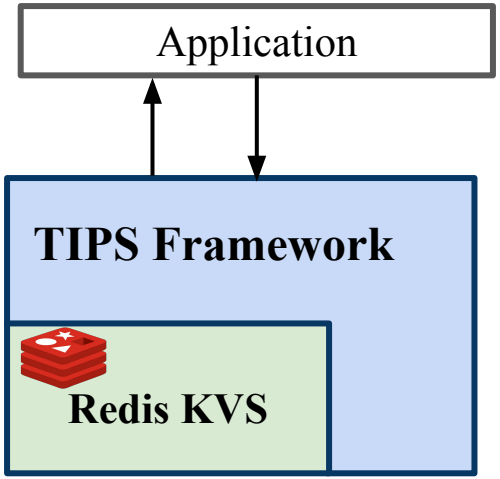
What if an application does not need transactional interface? Or How can we enable legacy KVS applications to work on NVM?

TIPS:** provides a **black-box framework** for legacy volatile **DRAM based KVS and index structures** to run on NVM by **transparently** making them **crash consistent and recoverable

- Space efficient **generic caching** technique designed for **high performance**
- **UNO logging** for efficient **crash consistency** and consistent **no-loss recovery**
- **Tiered concurrency model** with **adaptive scaling for** multi-core scalability and **durable linearizability**

TIPS: high level idea and contributions

*Application can **access Redis** KVS via the TIPS framework using the **facade APIs***



*Application can **plug-in** Redis or any other KVS using **plug-in APIs** provided by TIPS*

The plug-in process is agnostic i.e., same two steps for any application data structure and concurrency control

- Replace memory allocation to NVM
  malloc() → tips_alloc()
- Instrument all store operations with
  tips_undo_add API for crash consistency

*TIPS guarantees **durable linearizability** for all its conversions, the **gold standard correctness condition***

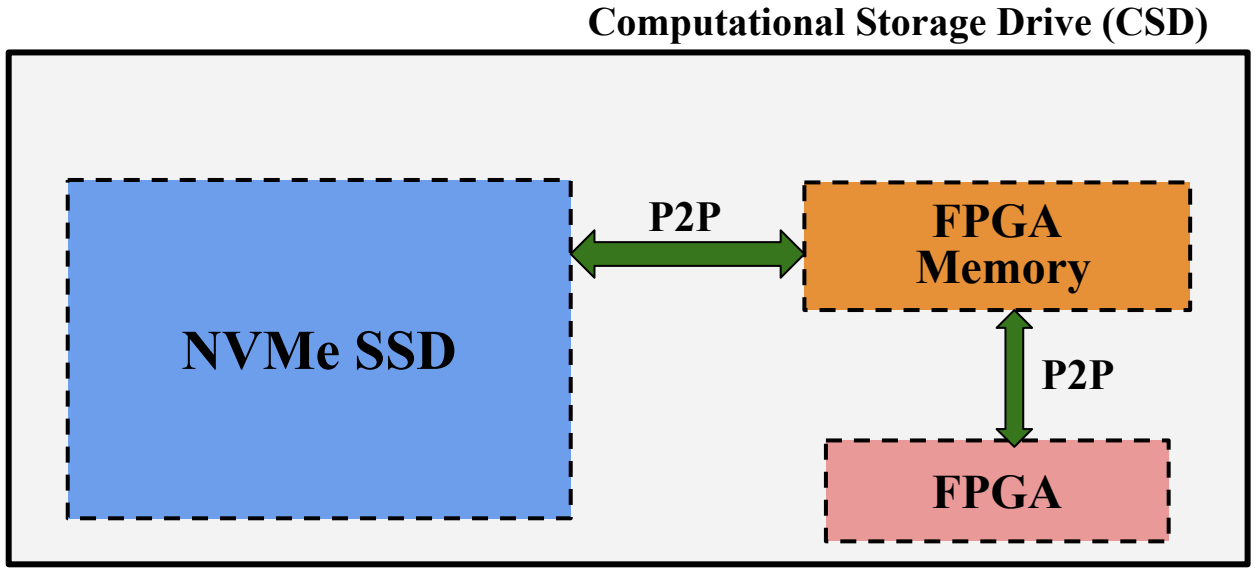
TIPS: Evaluation results

Indexes	Concurrency Control	LoC
Hash Table	Readers-Writer Lock	5/211
Lock-Free Hash Table	Non-blocking reads and writes	5/199
Binary Search Tree	Readers-Writer Lock	5/203
Lock-Free Binary Search Tree	Non-blocking reads and writes	5/194
B+Tree	Readers-Writer Lock	8/711
Adaptive Radix Tree	Non-blocking reads and blocking writes	9/1.5k
Cache-Line Extensible Hash Table	Non-blocking reads and blocking writes	8/2.8k
Redis Key-value Store	Blocking reads and writes	18/10k

Converted 8 applications with different with different concurrency control techniques, less than ***20 LoC changes for all the conversions***

TIPS enabled indexes supports critical features and performs, scales on-par or better *than indexes that are particularly optimized for NVM*

RETINA: KVS framework for scaling DL training using CSD

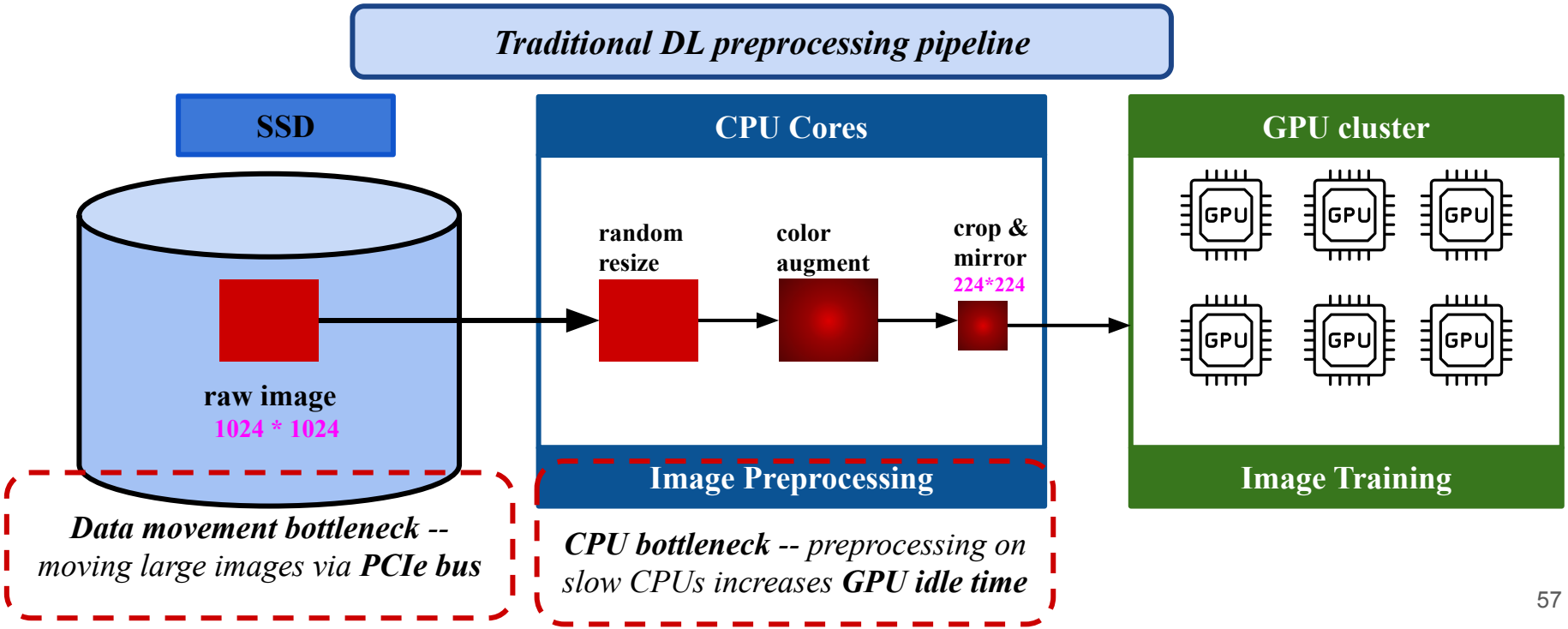


Near-storage accelerator performs ***compute on the SSD*** *data* ***without moving the data back and forth to the CPU memory***

CSD- Computational storage drives
DL - Deep Learning

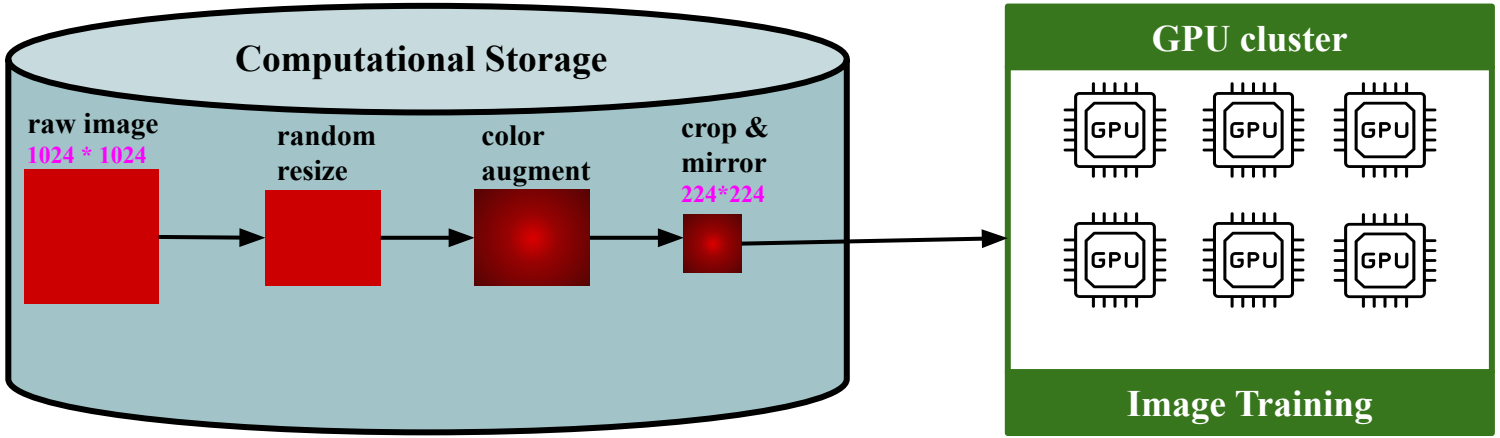
RETINA: KVS framework for scaling DL training using CSD

*Can we leverage **CSD** to solve the storage side **data movement bottleneck** and CPU side **preprocessing bottleneck** in the **Deep Learning (DL) training?***



RETINA: KVS framework for scaling DL training using CSD

DL preprocessing pipeline in RETINA



*Preprocessing is done **on the SSD** using the near-storage accelerator*

***Reduces CPU utilization --** host CPUs are free from performing preprocessing*

***Reduces the data movement --** Only the smaller preprocessed images are moved via the PCIe bus*

***Reduces GPU idle time --** faster preprocessing and reducing data movement improves the GPU utilization*

RETINA: KVS framework for scaling DL training using CSD

RETINA** exposes the preprocessing offload to storage using **KVS framework

- **KVS to index data on the SSD** -- lookup, scan, insert, delete operations
- **Unified preprocessing pipeline** on the **near-storage FPGA** -- perform preloaded preprocessing functions

*Integrated **RETINA to the Tensorflow** and performed **ResNet50 DL model training***

- CPU utilization **reduced by 97%** and GPU utilization **improved by 25%** as compared to the traditional pipeline model
- **36% faster preprocessing on the near-storage FPGA** than preprocessing images **on the cpu** in the traditional pipeline model

Other Research Works

- Many-core scalable synchronization primitive for DRAM
- MV-RLU scales up to 440 CPU cores

➢ **MV-RLU: Scaling Read-Log-Update with Multi-Versioning**

Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, R. Madhava Krishnan, and Changwoo Min ***(ASPLOS 2019)***

- Scalability and Memory Safety from the NVM allocator POV
- Uncovers the safety vulnerabilities in the PMDK allocator

➢ **POSEIDON: Safe, Fast and Scalable Persistent Memory Allocator**

Anthony Demeri, Wook-Hee Kim, R. Madhava Krishnan, Jaeho Kim, Mohannad Ismail, and Changwoo Min
(Middleware 2020)

- Design principles for NVM software design
- Tackles scalability across NUMA, concurrency and crash consistency

➢ **PACTree: A High Performance Persistent Range Index Using PAC Guidelines**

Wook-Hee Kim, R.Madhava Krishnan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min ***(SOSP 2021)***

Talk Summary: Solution contributions

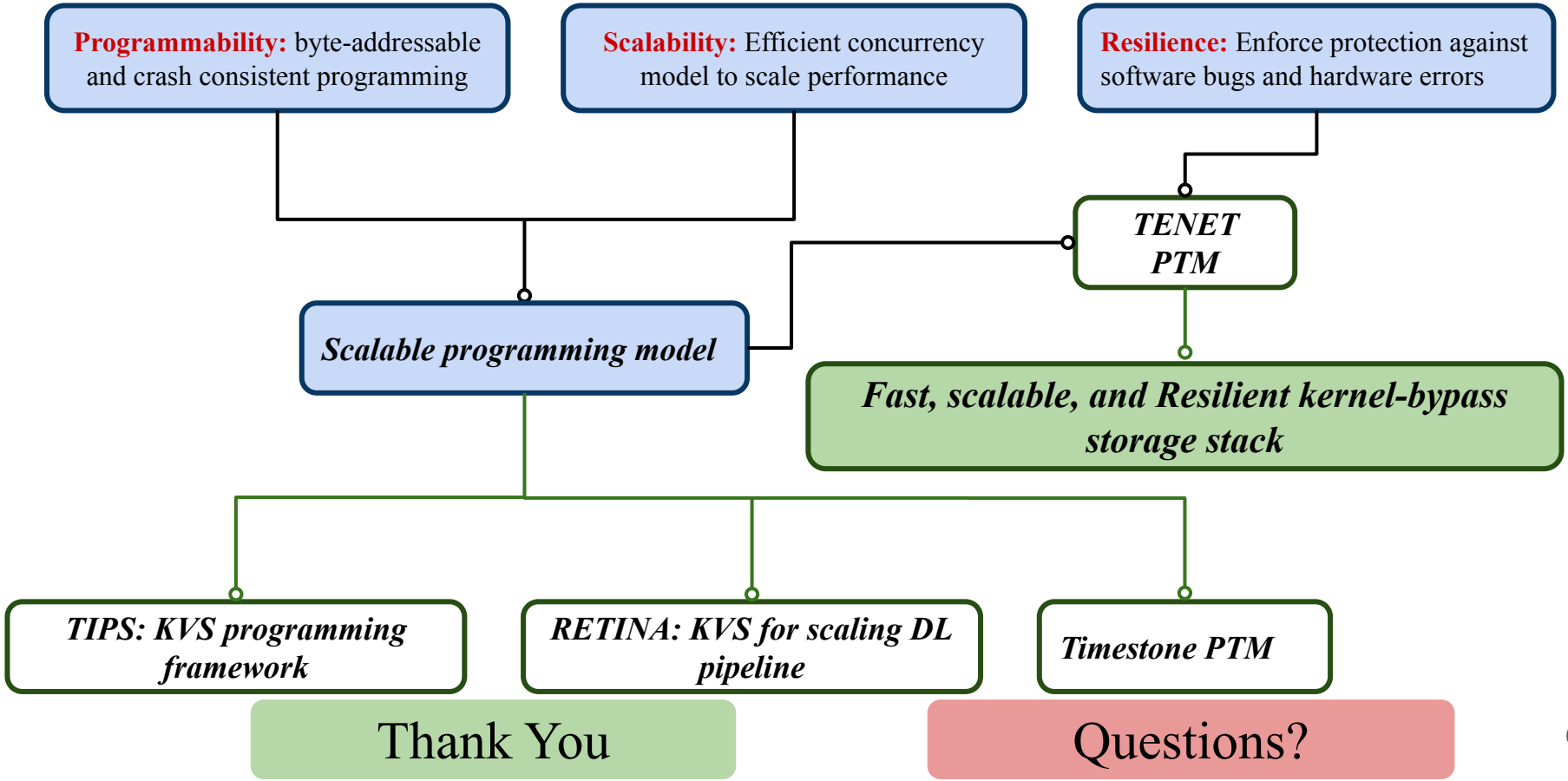
Feature	OS storage stack	Kernel-bypass storage stack [TENET]
Programming abstraction	**Filesystem**	Persistent Transaction
Programming support	**No, only block-adderssable**	Yes, byte and block-addressable
Multi-core scalability	**Poorly scalable**	Scales across 100's CPU cores
Latency scale	**Micro-second scale**	Nano-second scale
Isolation and Safety	**Yes**	Yes, also fault tolerant

Why transactional programming abstraction?
- ***Makes programming easy***
- ***Hides concurrency and crash consistency***

How to make transactional programming scalable and high-performant?
- ***MVCC for Scalability***
- ***TOC Logging for high performance***

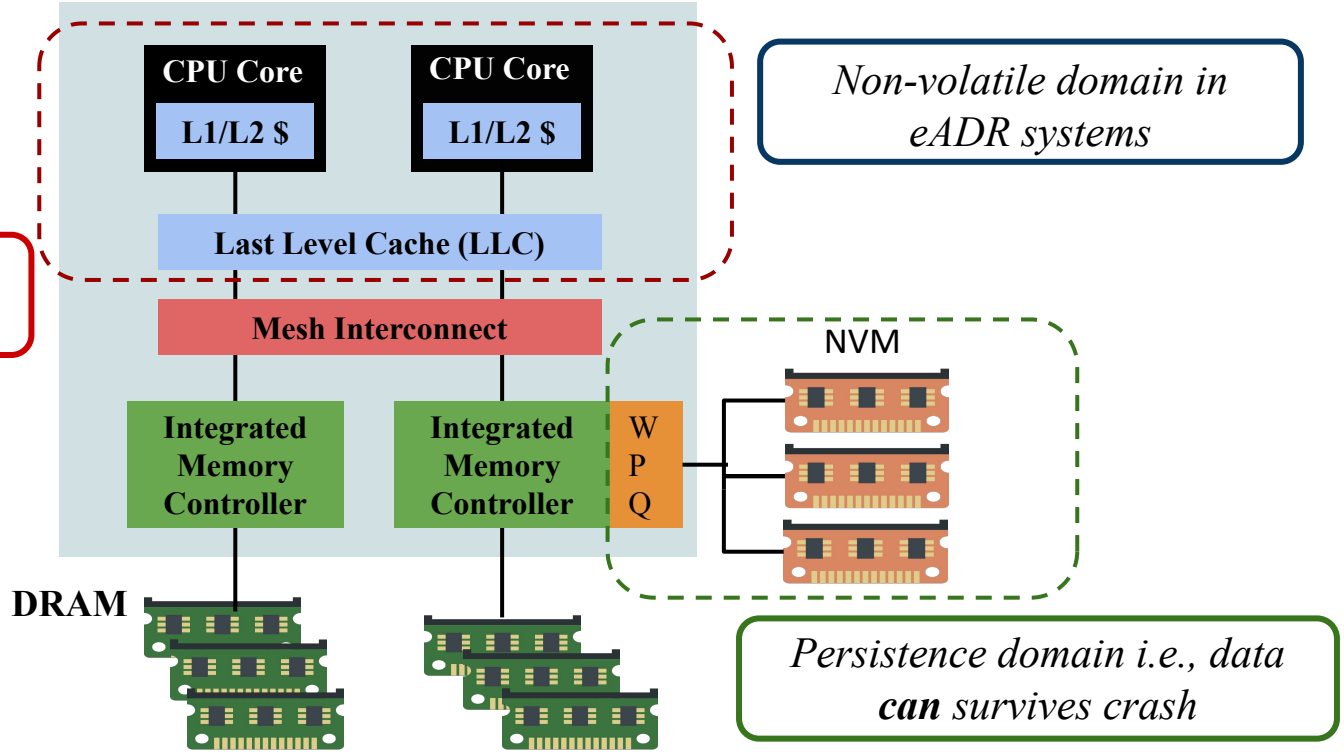
How to make transactional programming memory safe and fault tolerant?
- ***MPK and Canary bits for spatial memory safety; pointer tags for temporal memory safety***
- ***Asynchronous NVM-SSD replication for performance and cost-efficient fault tolerance***

Talk Summary: Achieving the research vision



Backup

A closer look at the byte-addressable NVM



Non-volatile domain in eADR systems

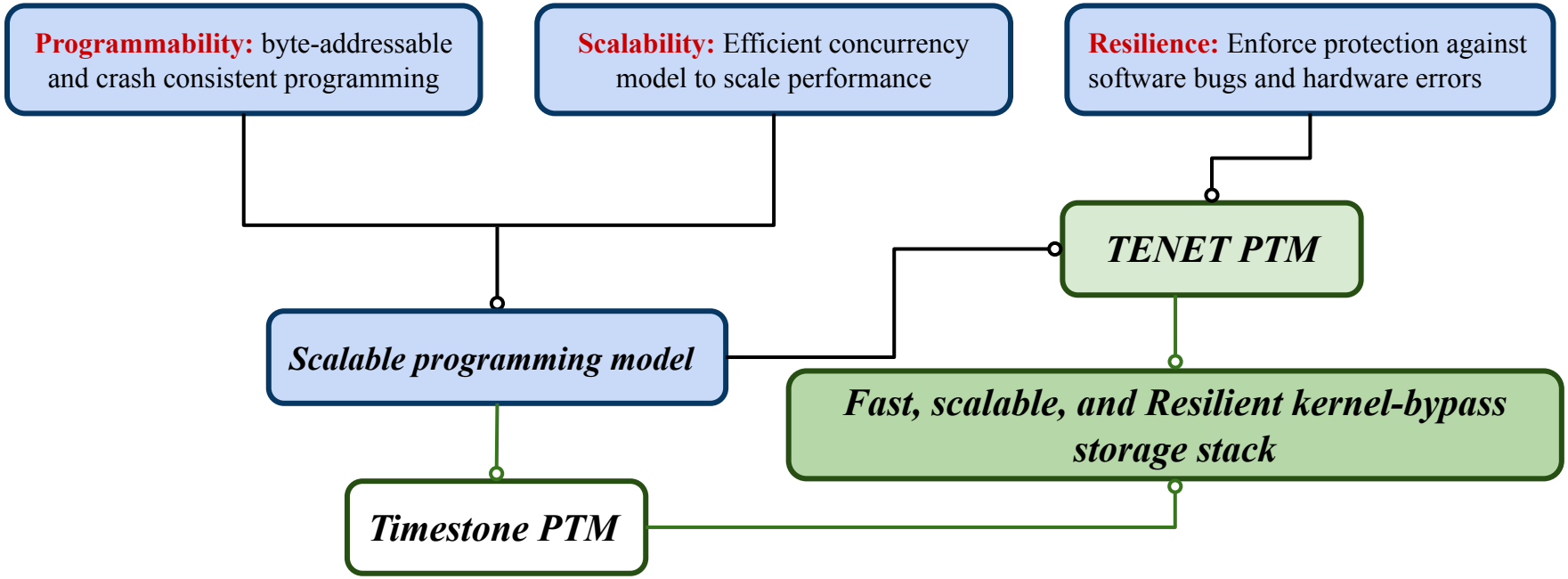
*Volatile domain i.e., data **can not** survive a crash*

*Persistence domain i.e., data **can** survives crash*

How can applications leverage the NVM's byte-addressability feature?

WPQ: Write Pending Queue

Talk Summary: Achieving the research vision



Thank You

Questions?

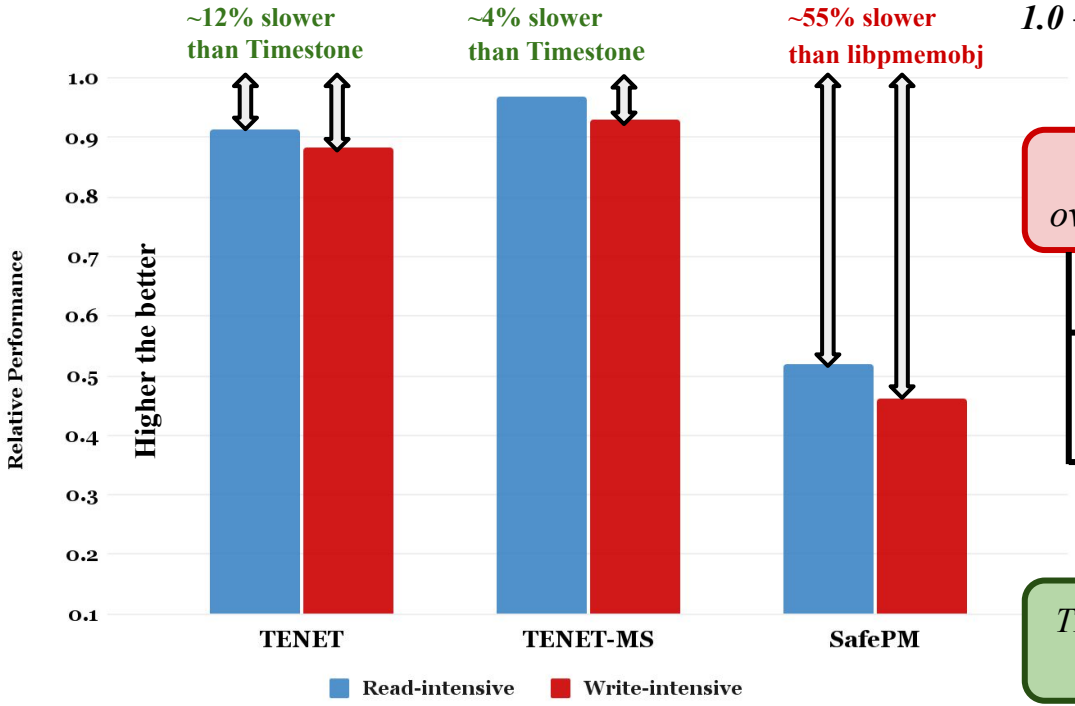
Future works

Rethinking the techniques in the context of CXL enabled storage stack *-- byte-addressable programming, scalability, and memory safety problems are highly relevant*

Concurrent programming framework for CXL enabled cache coherent accelerators *-- extending Timestone PTM to support transactional programming across accelerators and CPUs*

Extending RETINA to support programming other CSD architectures -- *programming computational storage array would enable RETINA to scale distributed DL training*

Performance of TENET and SafePM for hash table



*SafePM incurs ~**55%** performance overhead for only memory safety guarantee*

- *Checks **every NVM loads and stores** -- high performance overhead*
- *Centralized memory safety metadata -- high crash consistency overhead*

*TENET adds ~**12%** performance overhead for memory safety and fault tolerance*

- Performance is normalized to their respective baseline PTMs
  - **SafePM normalized to the libpmemobj → throughput (safePM)/throughput (libpmemobj)**
  - **TENET normalized to the TimeStone → throughput (TENET)/throughput (Timestone)**

Background on types of Media Errors

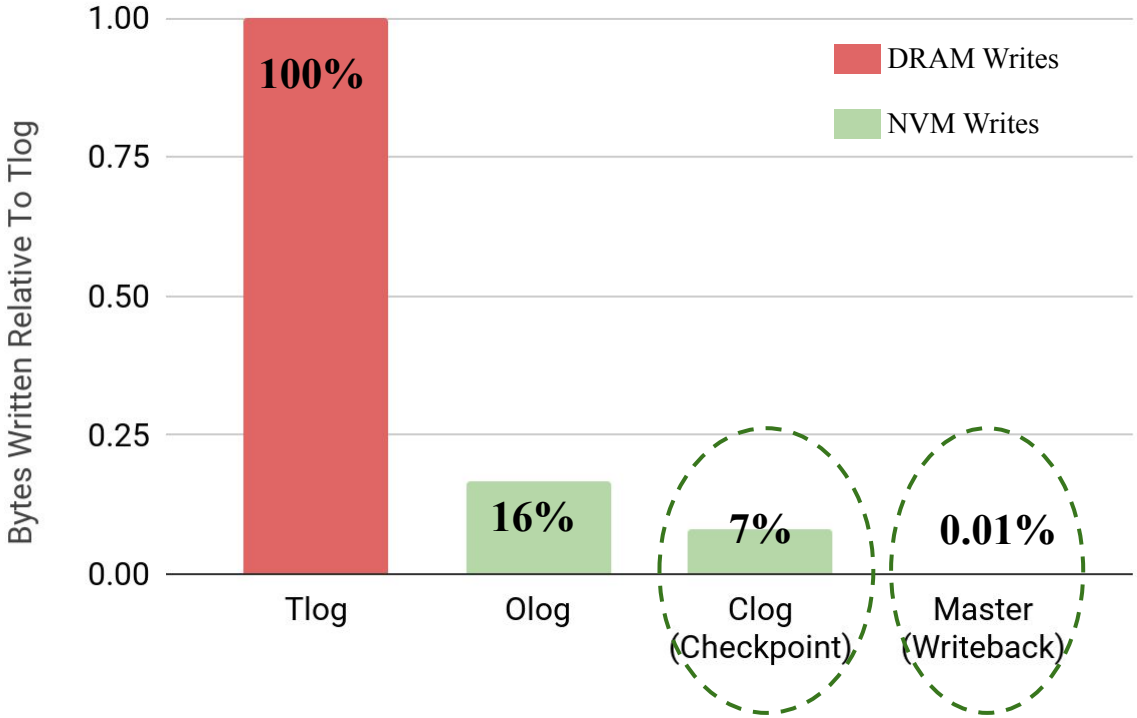
02 **NVM Media Errors**

- Correctable Media Errors
- **Uncorrectable Media Errors**

- NVM has **high Random Bit Error Rate (RBER)** ~= NAND flash
- **Uncorrectable media errors (UME) are detected by the hardware ECC but can not be corrected**
  - UME can happen at random offset and the OS kernel offlines the corrupted NVM page
  - **Application is responsible for fixing the corrupted NVM page**

Applications are required to maintain a backup of NVM data to rollback the affected NVM page to prevent data loss

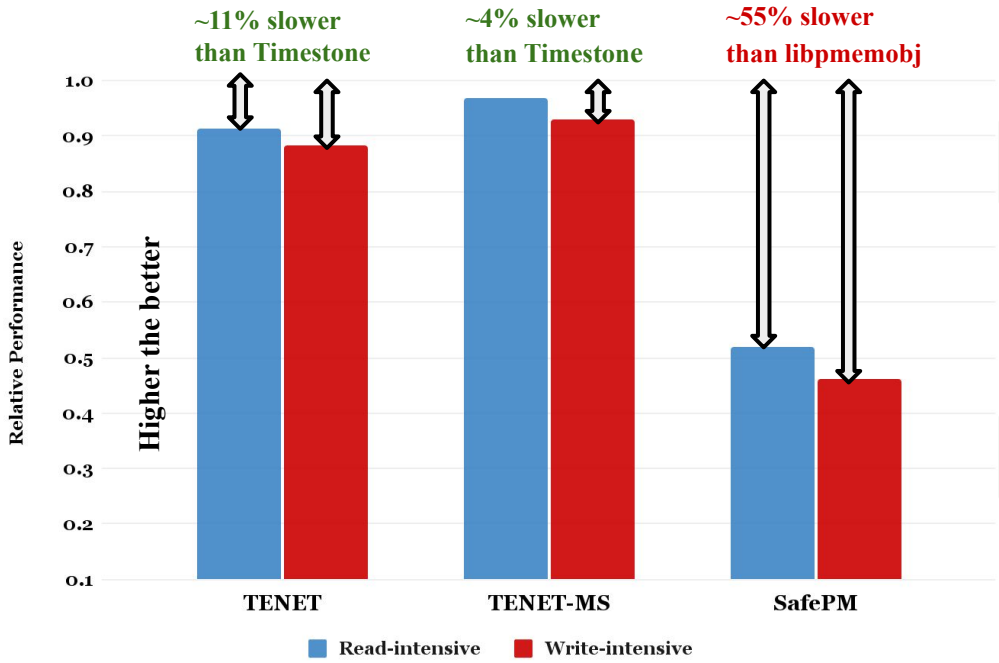
Write Coalescing in TOC Logging



- Only 7% of writes are checkpointed from Tlog
- The rest are coalesced in the Tlog

- Only 0.01% of writes are written back to master
- The rest are coalesced in the Tlog and Clog

Performance of TENET and SafePM for hash table



- Performance is normalized to their respective baseline PTMs
  - **SafePM normalized to the libpmemobj**
  - **TENET normalized to the TimeStone**

TENET does not require additional crash consistency operations for its memory safety metadata

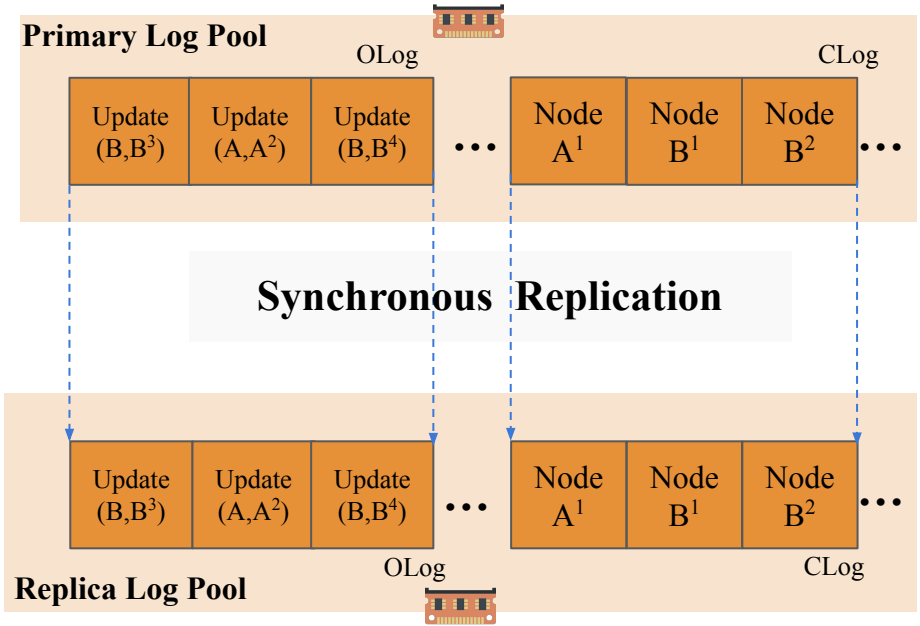
- MPK → hardware primitive
- Pointer tags → embedded directly into the object

TENET does not perform memory safety validation for every NVM access

- Spatial safety checks performed only at the commit time
- Temporal safety checks performed only at the first-dereference of an NVM object

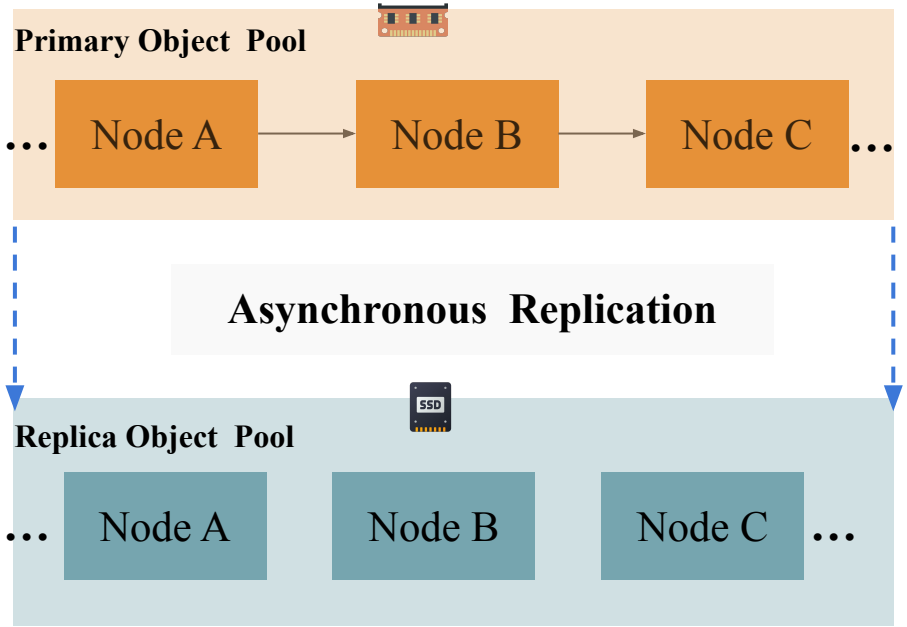
Replicating NVM data for fault tolerance

NVM Log Pool Replication



The logs are synchronously replicated on the NVM in the critical path

NVM Object Pool Replication



The NVM objects are asynchronously replicated on the SSD out of the critical path