Programming Interfaces to Non-Volatile Memory

Michael Swift

University of Wisconsin—Madison

Outline

- NVM solid-state drives
- Persistent memory file systems
- Persistent Regions
- Persistent data stores

SNIA Model

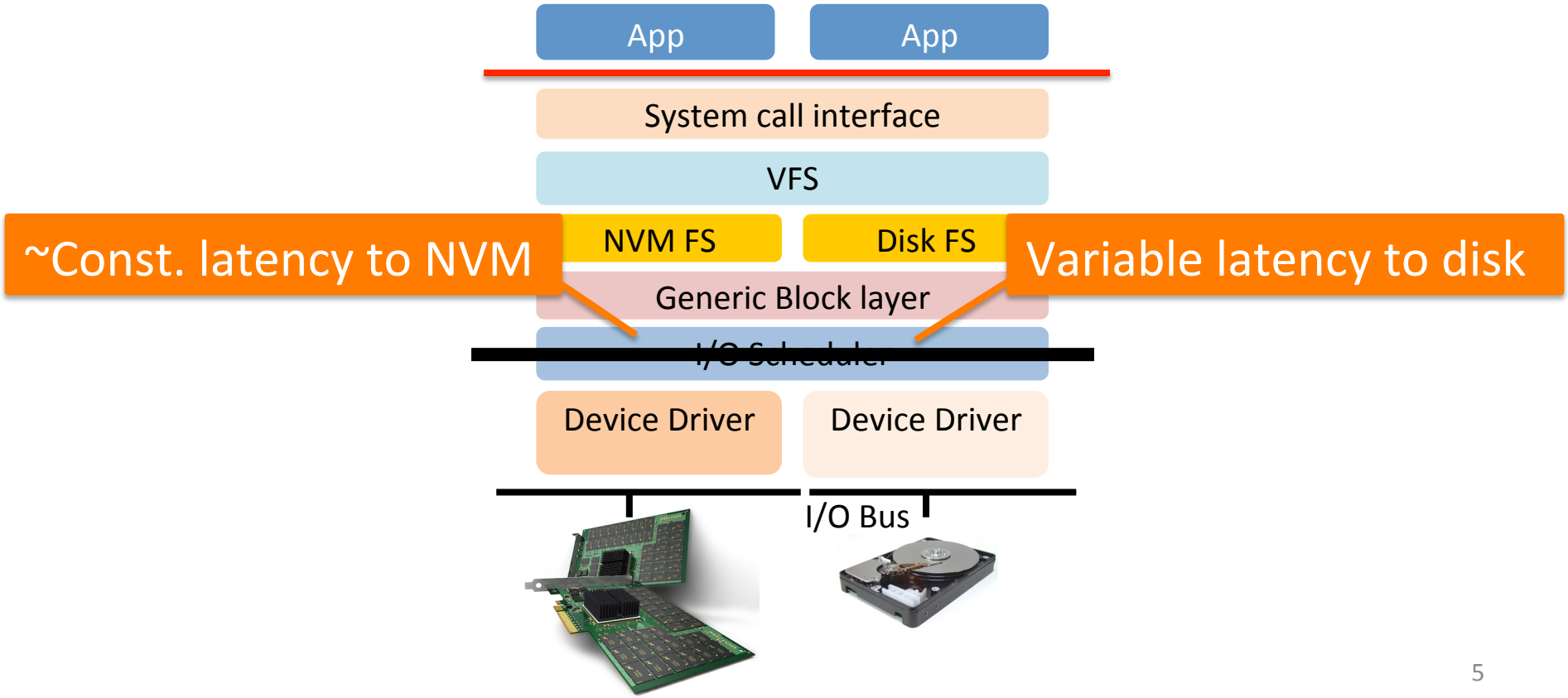
- NVM block mode: NVM-attached devices
  - Legacy or enhanced file system
- Persistent Memory Volume: RAM disk
  - Raw access to PM from kernel
  - Legacy file system
- Persistent Memory File System: PMFS, SCMFS, BPFS
  - File system tuned for NVM properties
- Persistent memory: NV-Heaps, Mnemosyne, CDDS
  - Load-store access from user mode

[Rudoff ;login'13]

Modes of NVM Use

Mode	Programming Interface	NVM Accessibility
NVM block	Read/write block	Kernel, User
PM Block	Read/write block, byte	Kernel
PM File	Open, read, write file	Kernel
PM	Load, store	User

NVM as SSD

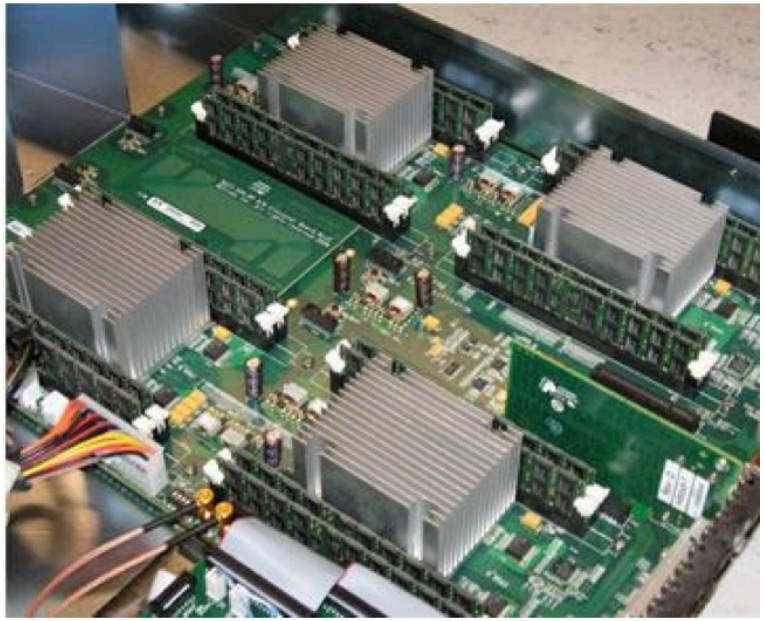

App
App
System call interface
VFS
NVM FS
Disk FS
Generic Block layer
I/O Scheduler
Device Driver
Device Driver
I/O Bus
~Const. latency to NVM
Variable latency to disk

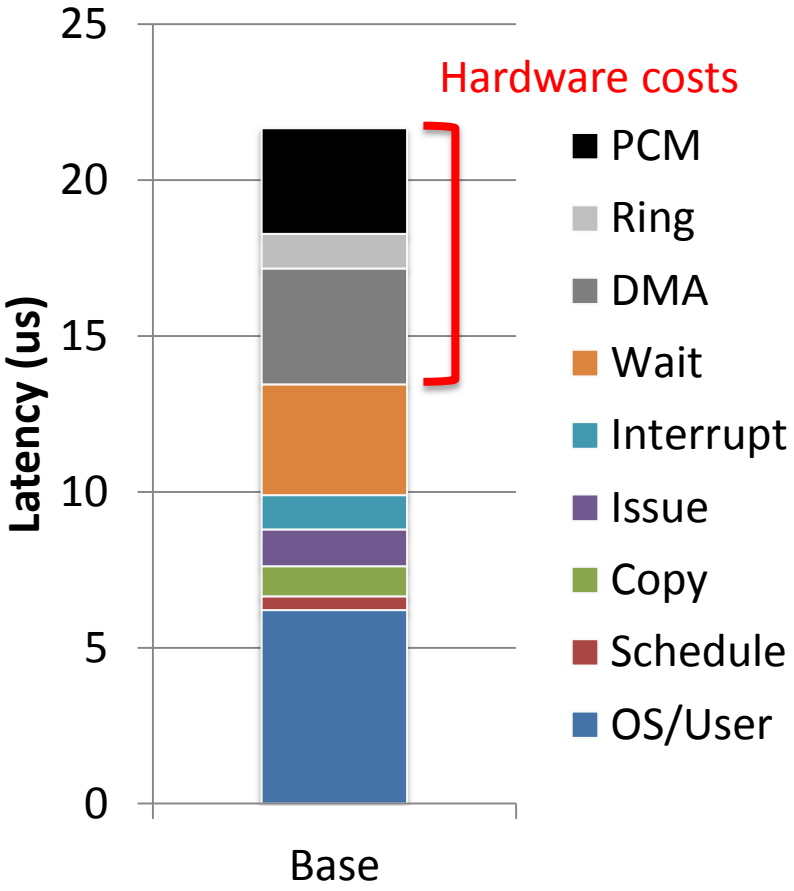
Devices: NVM-based SSDs

- Challenges
  - Hardware interface
  - Software latency
  - Protection and user-mode access
- Examples:
  - Moneta-D
  - NVMExpress

NVM SSD Challenges

- Software overheads in kernel



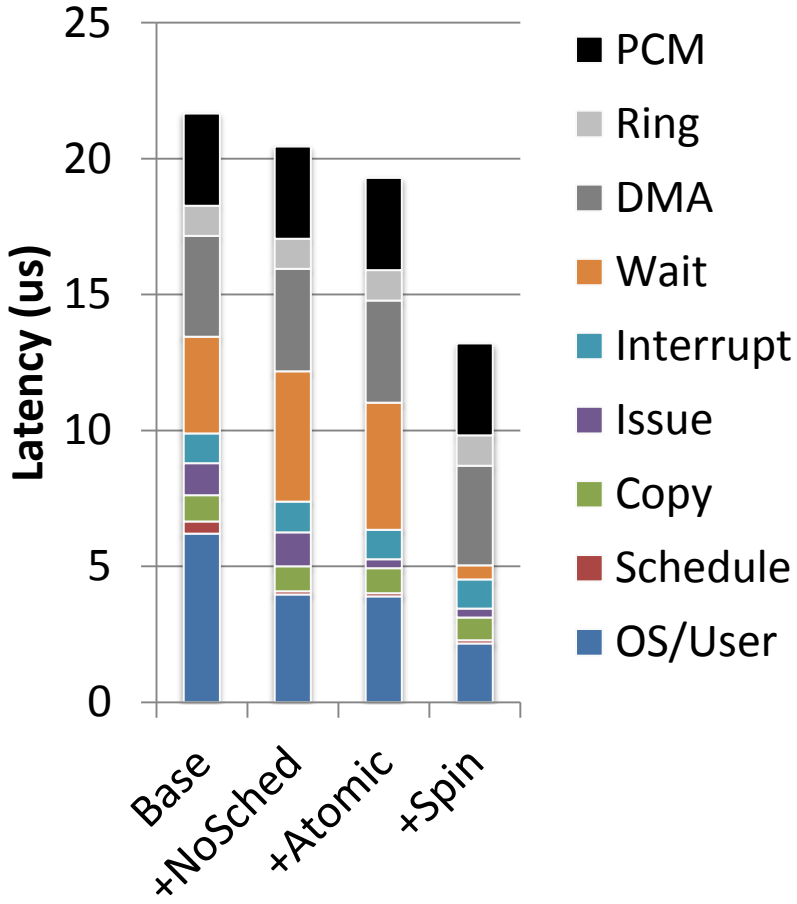


[Caulfield, SC'10]

OS Scheduling Overhead

- I/O scheduling
- Locking
- Interrupts and waiting

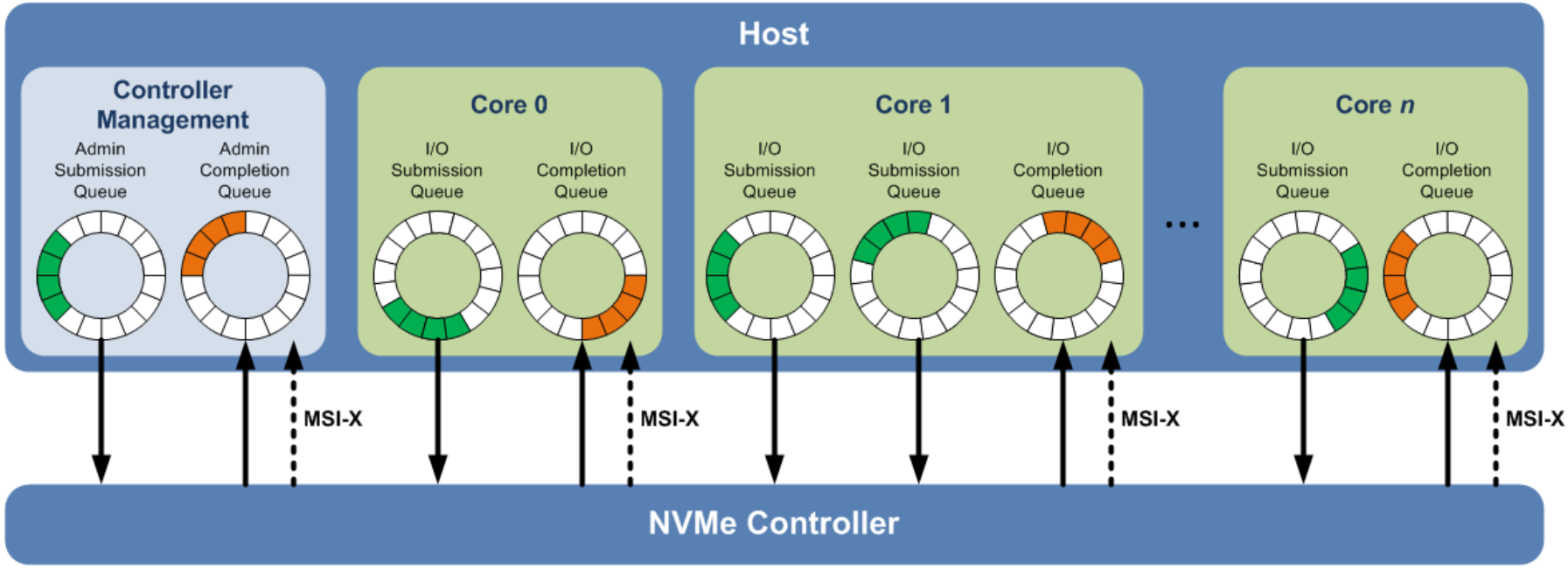
- Example: Moneta [MICRO'10]



[Caulfield, SC'10]

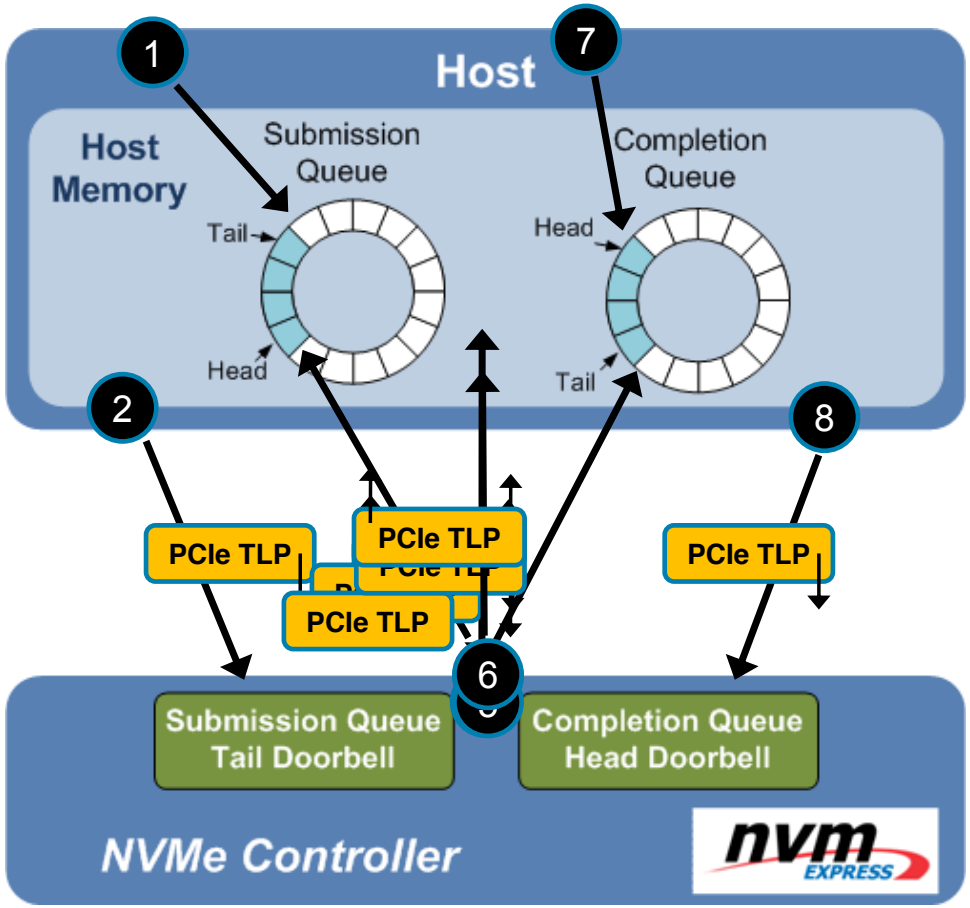
NVM Express

- All parameters for 4KB command in single 64B DMA fetch
- Deep queues (64K commands /queue, u64K queues)
- 3 µs latency: no cached I/O reads



Intel IDF ‘12

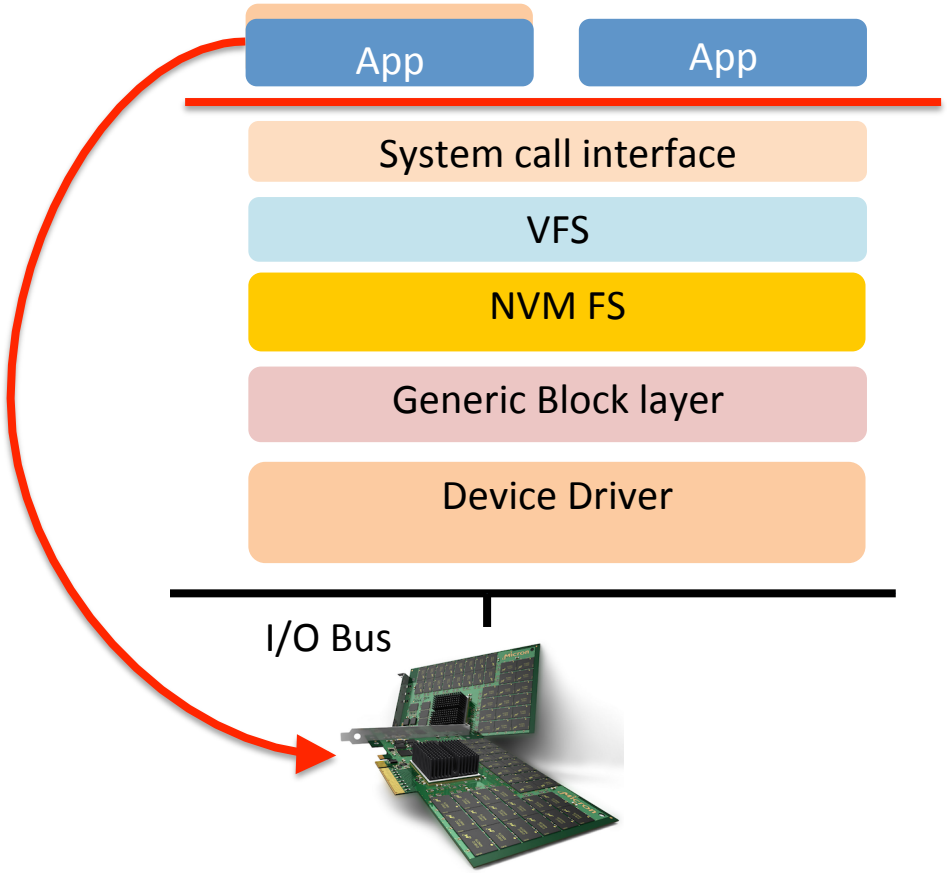
NVMexpress Operation



1) Queue Command(s)
2) Ring Doorbell *(New Tail)*
3) Fetch Command(s)
4) Process Command
5) Queue Completion(s)
6) Generate Interrupt
7) Process Completion
8) Ring Doorbell *(New Head)*

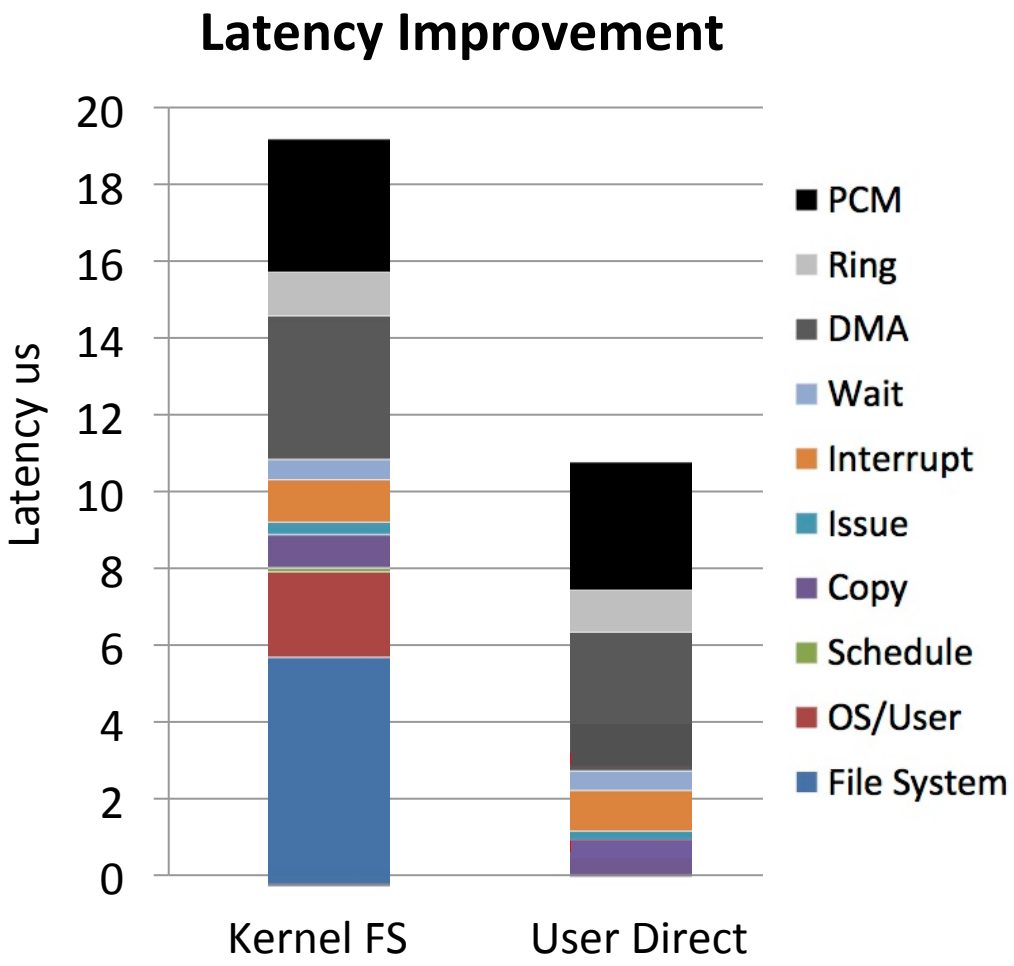
User-mode Data Access

- Challenge: metadata, I/O Latency



Remove FS Overhead

- Metadata calls still in kernel
- Kernel grants program access to **data**
- Program issues DMA directly to device
- Example: Moneta-D [ASPLOS'12]

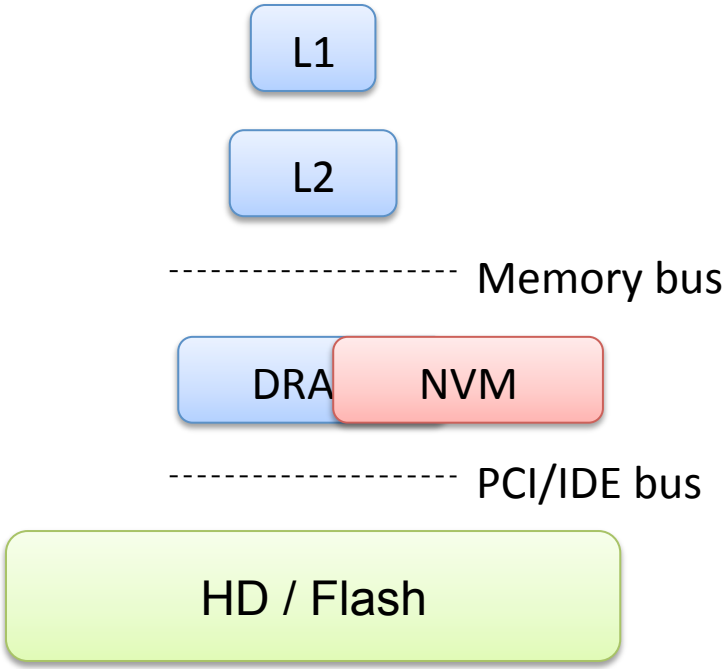


[Caulfield, ASPLOS'12]

Outline

- NVM solid-state drives
- Persistent memory file systems
- Persistent Regions
- Persistent data stores

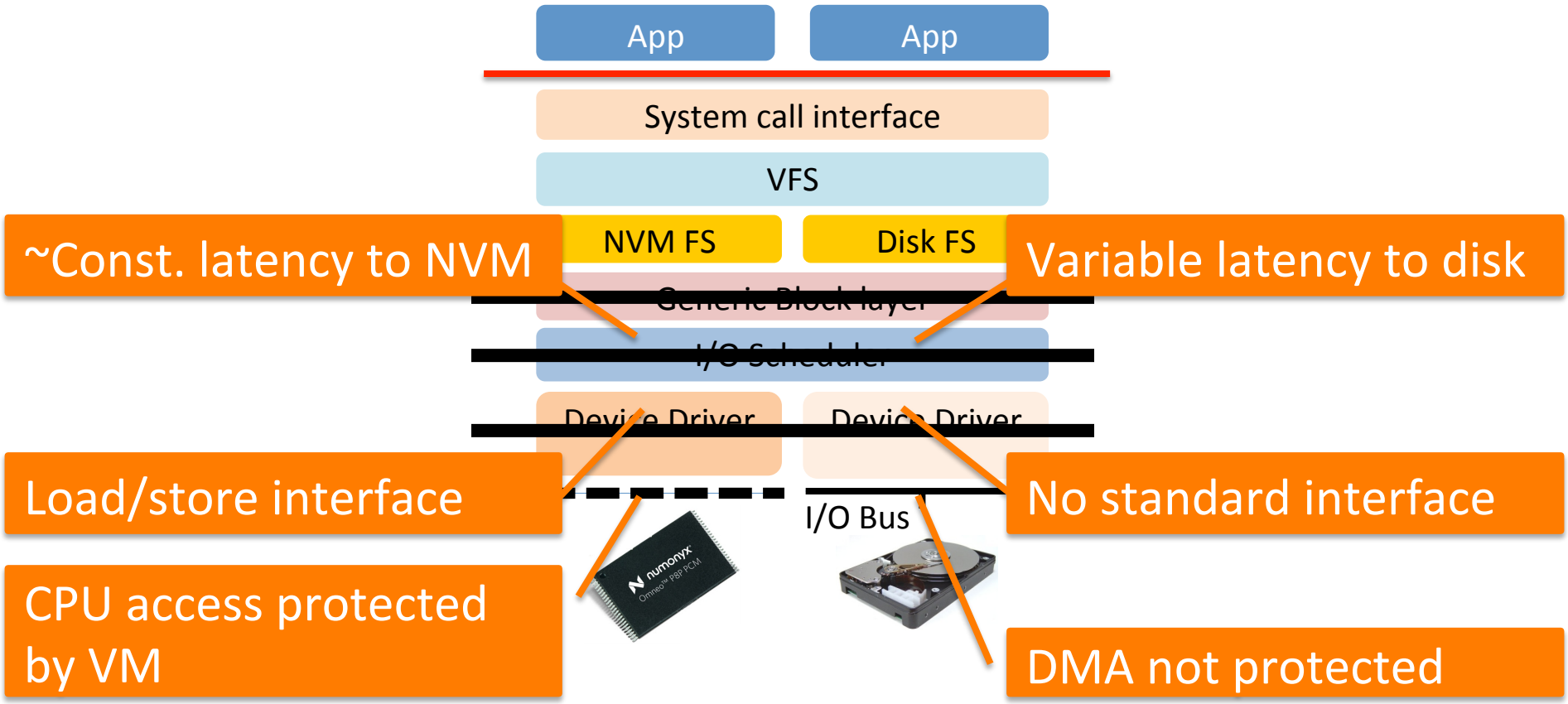
NVM in the PC



- NVM and DRAM are **addressable** by the CPU
- Physical address space is **partitioned**
- NVM data may be **cached** in L1/L2

[Condit, SOSP'09]

Direct VS Peripheral Access



Option 1: NVM Disks

- Design:
  - Small block driver exposes regions of NVM as a block devices
  - Use unmodifed file systems
- Issues:
  - Performance
- Example:
  - Light VM

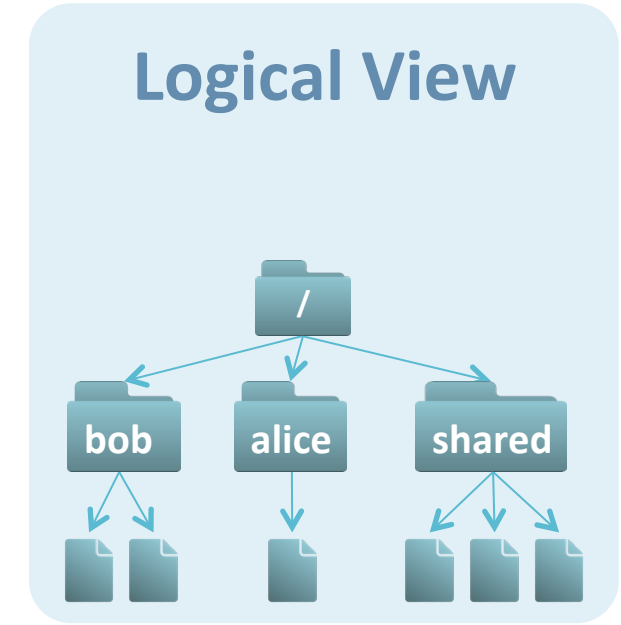
Option 2: NVM File systems

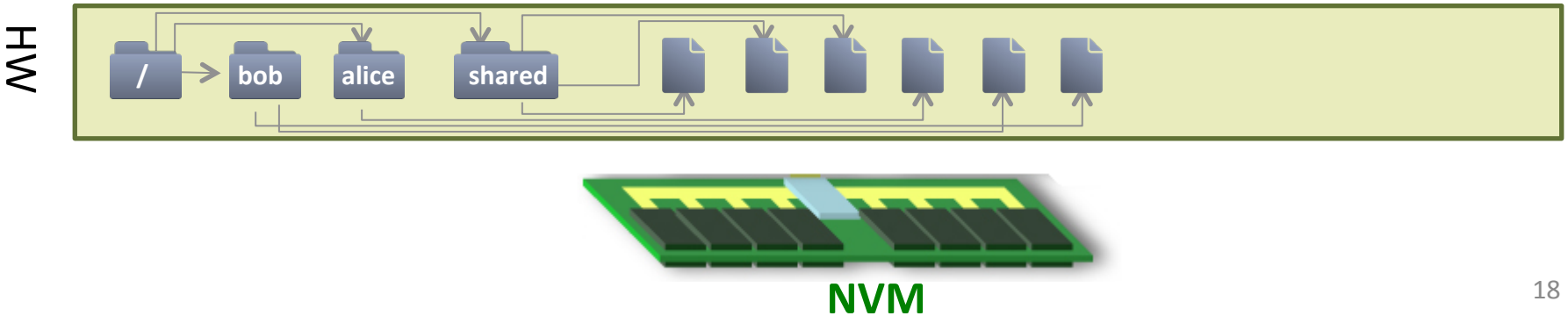
- Issues:
  - Consistency and Durability
  - Memory copies
- BPFS
- PMFS
- Aerie

Memory-Mapped File System

- All file data available through memory
- I/O operations become loads/stores

- Example: BPFS from MSR [SOSP'09]



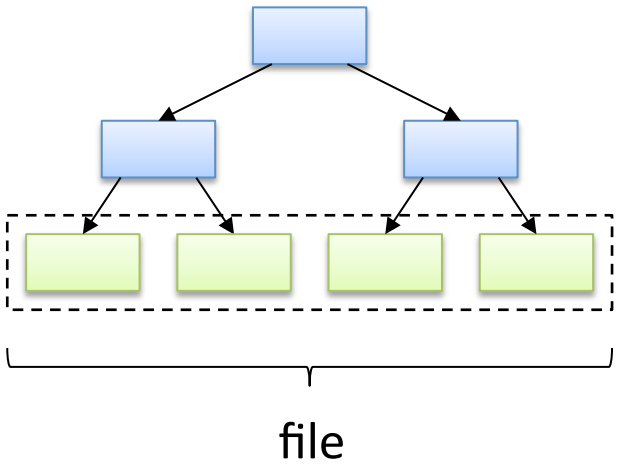


BPFS: A NVM File System

- New feature: Guarantees that all file operations execute atomically and in program order
- Better performance than disk-based on NVM disk
- Short-circuit shadow paging often allows atomic, in-place updates

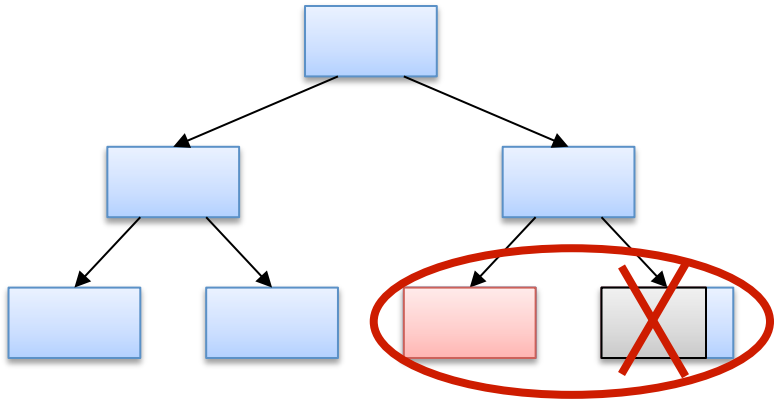
[Condit, SOSP'09]

BPFS: A NVM File System



Enforcing FS Consistency Guarantees

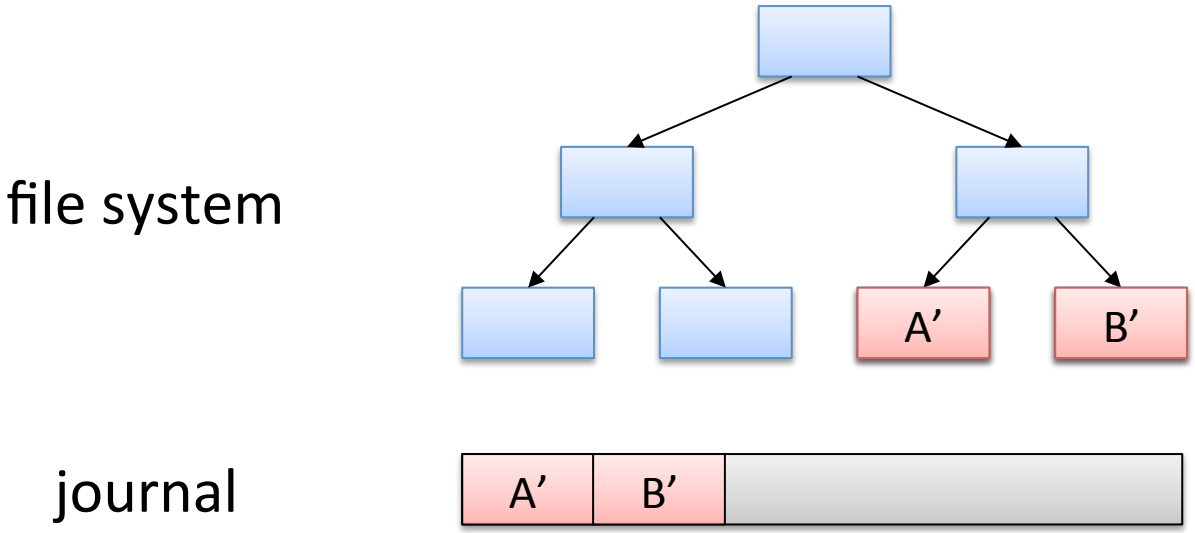
- What happens if we crash during an update?



  - Disk: Use journaling or shadow paging
  - NVM: Use short-circuit shadow paging

Review 1: Journaling

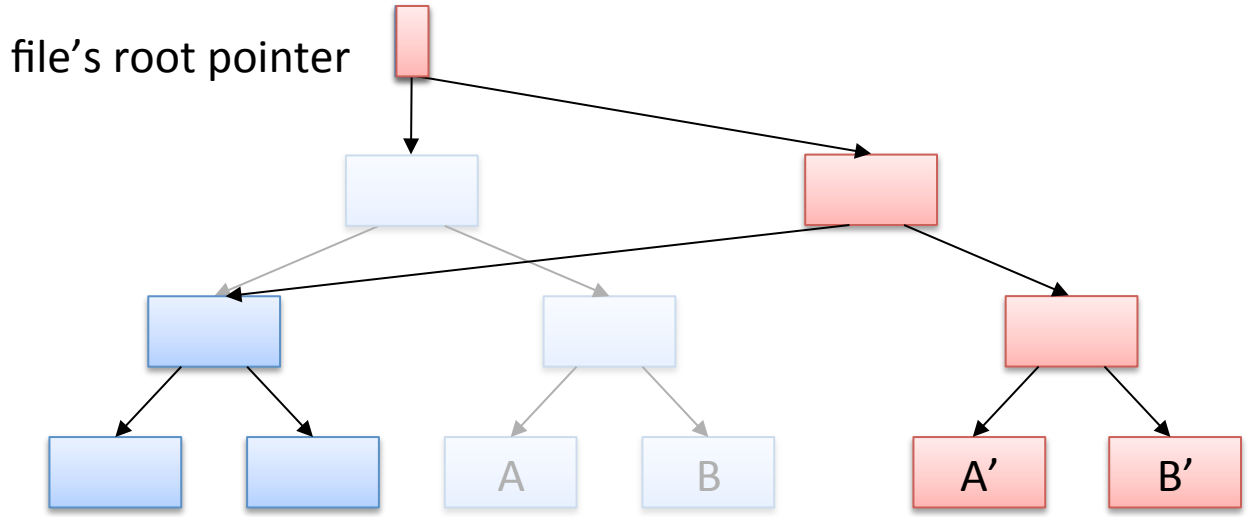
- Write to journal, then write to file system



- Reliable, but all data is written twice

Review 2: Shadow Paging

- Use copy-on-write up to root of file system



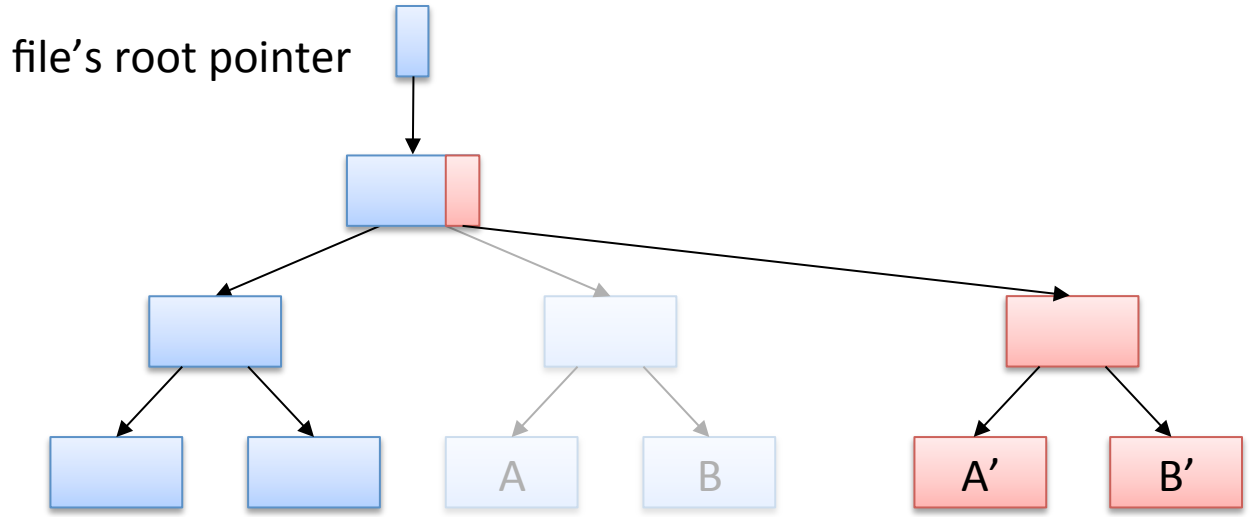
- Any change requires bubbling to the FS root
- Small writes require large copying overhead

Short-Circuit Shadow Paging

- Inspired by shadow paging
  - Optimization: In-place update when possible

Short-Circuit Shadow Paging

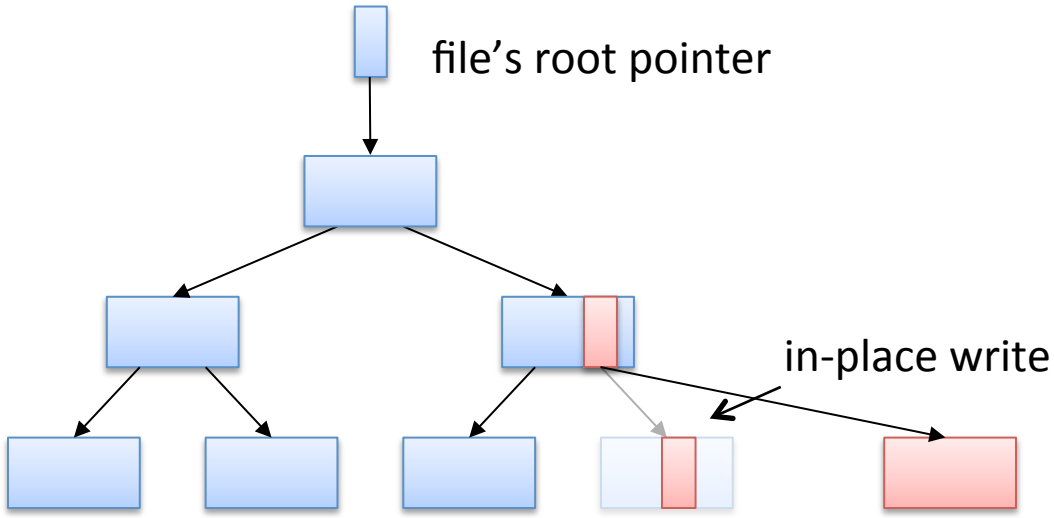
- Inspired by shadow paging
  - Optimization: In-place update when possible



- Uses byte-addressability and atomic 64b writes

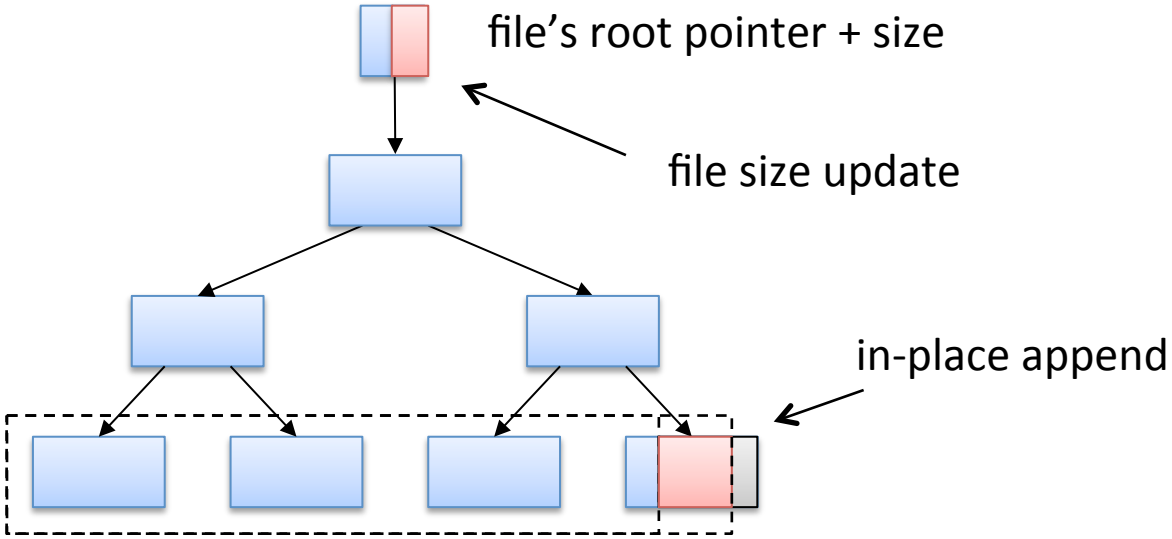
Opt. 1: In-Place Writes

- Aligned 64-bit writes are performed in place
  - Data and metadata

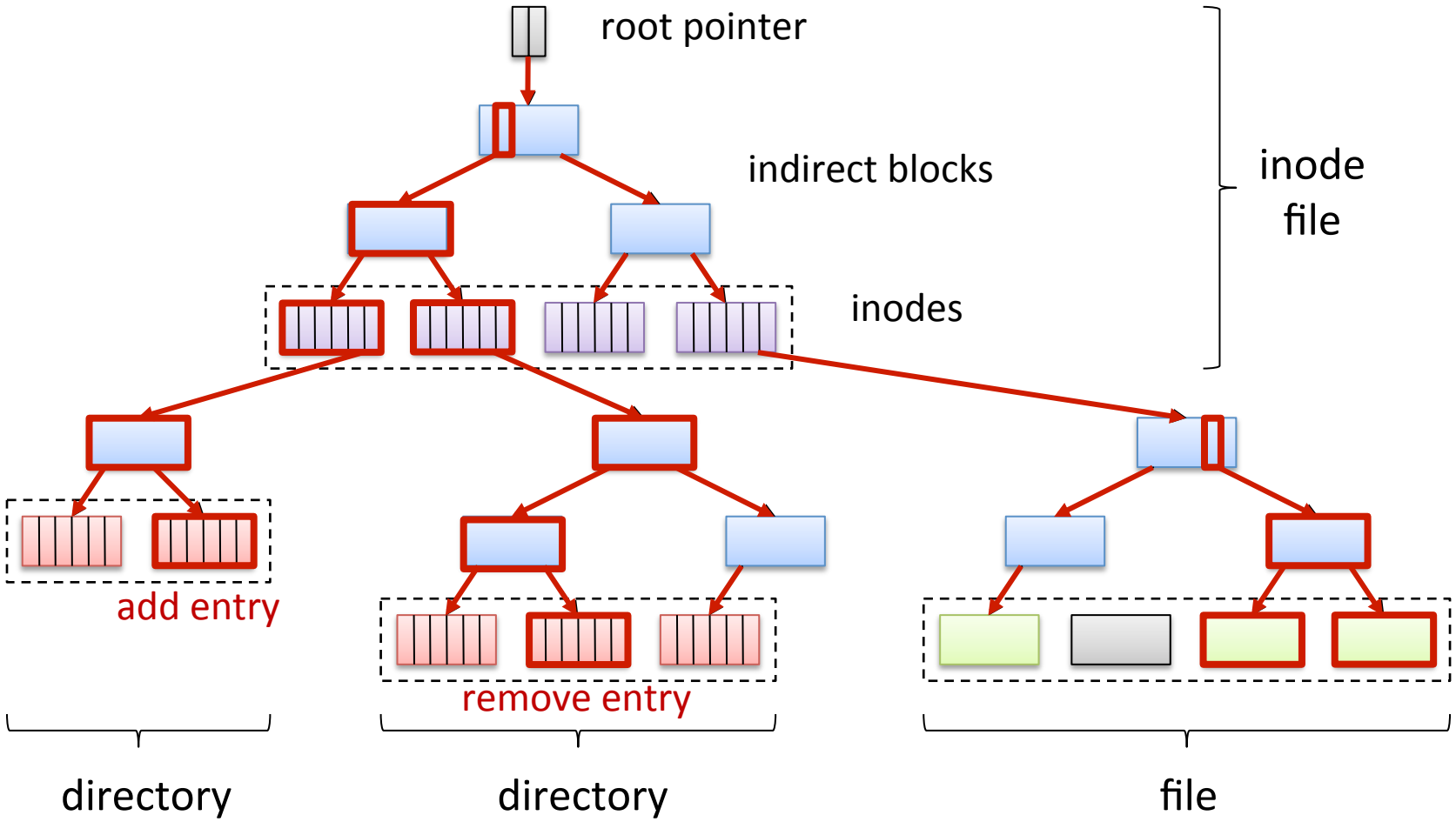


Opt. 2: Exploit Data-Metadata Invariants

- Appends committed by updating file size



BPFS Example

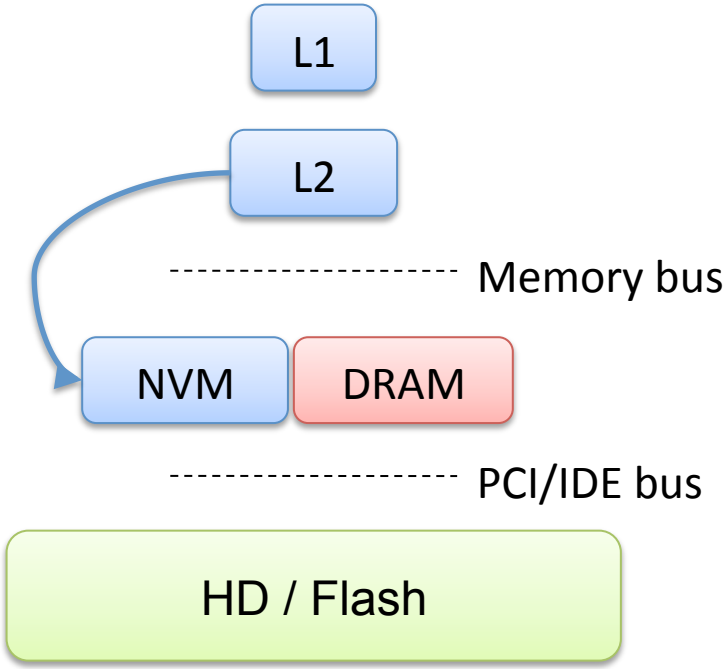


- Cross-directory rename bubbles to common ancestor

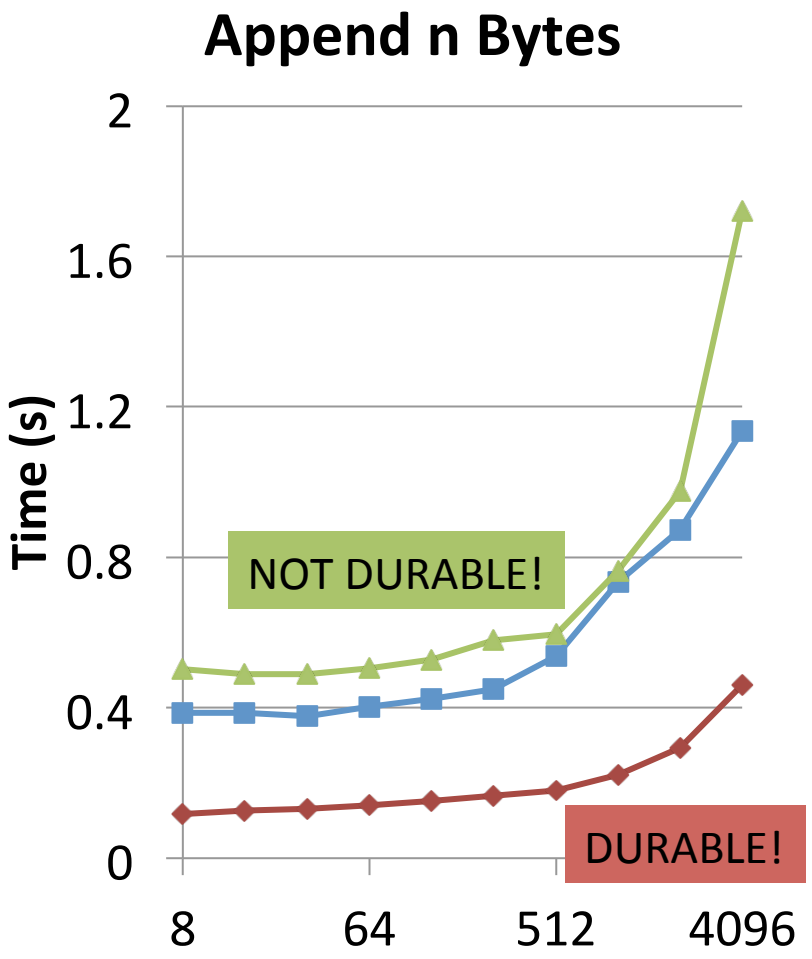
Cache Optimization

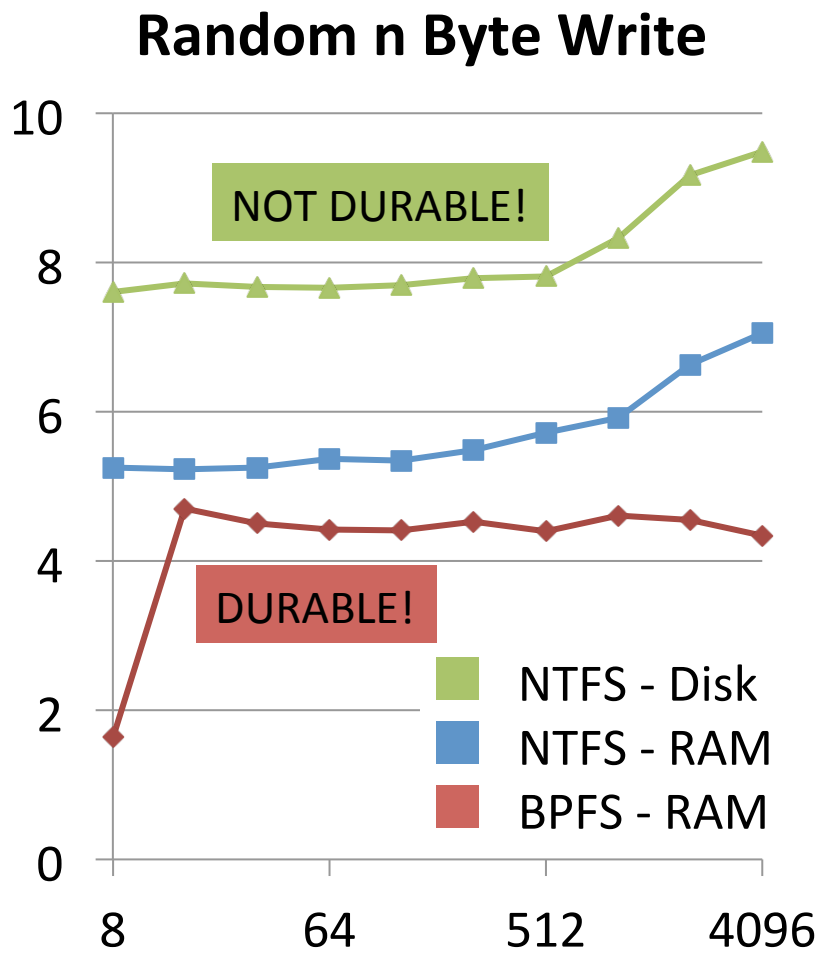
- Replace DRAM cache with processor L1/L2 cache
- Data persists as soon as it hits NVM

➢ Need to control order of cache evictions

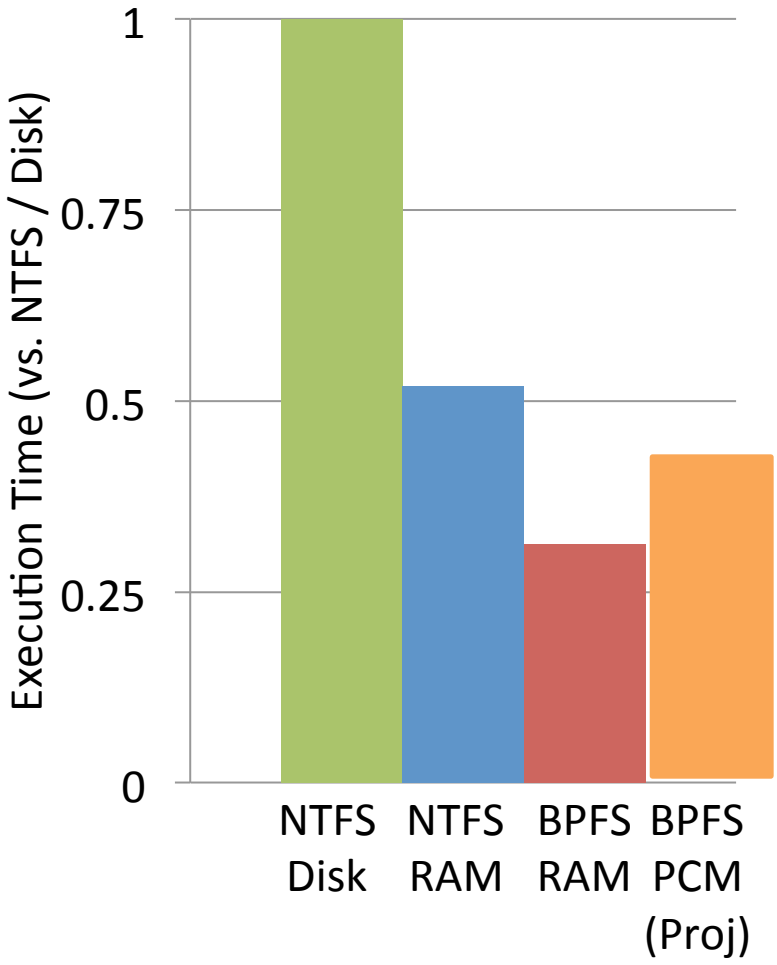


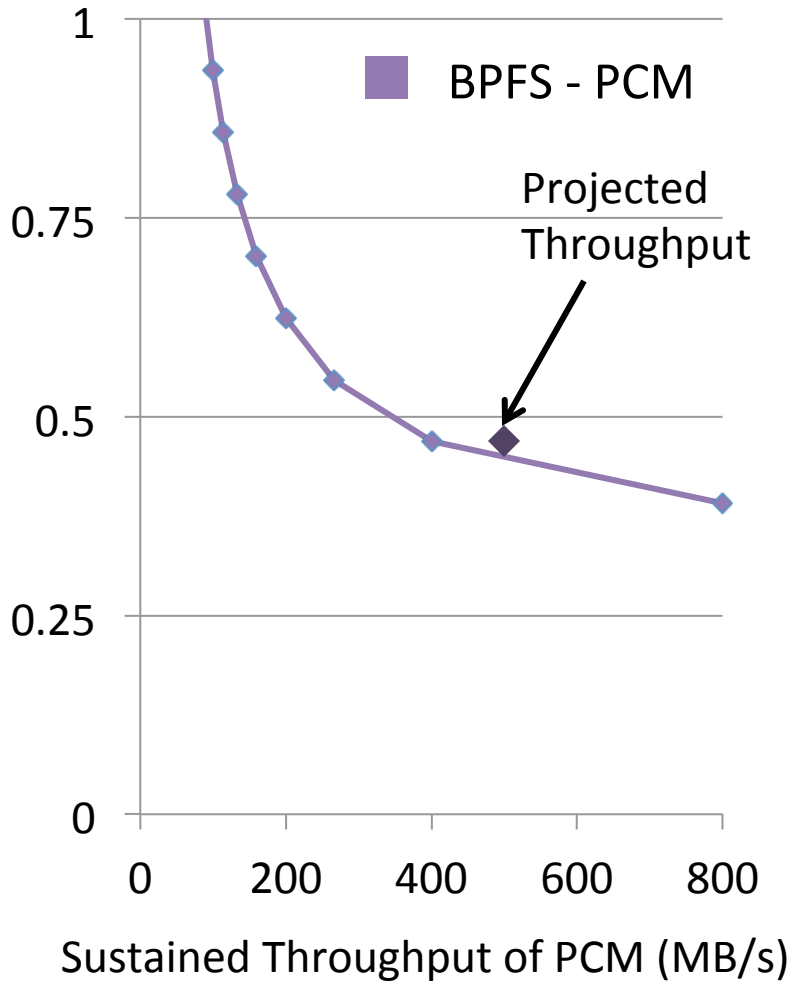
Microbenchmarks





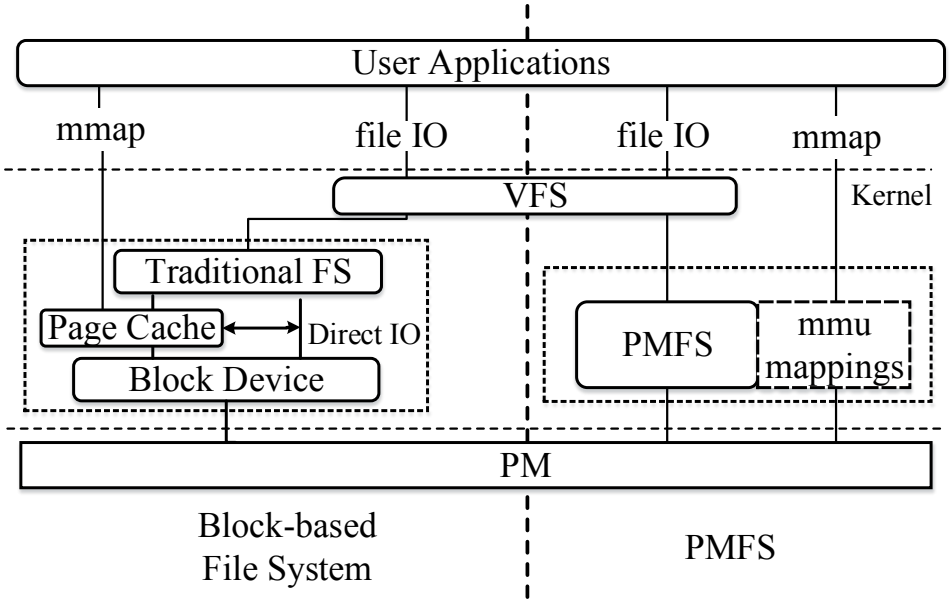
BPFS Throughput On PCM





PMFS – Linux DAX/XIP

- Lightweight Linux file system
- Leverages Direct Access (DAX) in Linux
  - Bypasses caching: copies data directly between user-buffers and NVM
  - NVM pages mmap()ed to user processes



[Kumar, EuroSys'14]

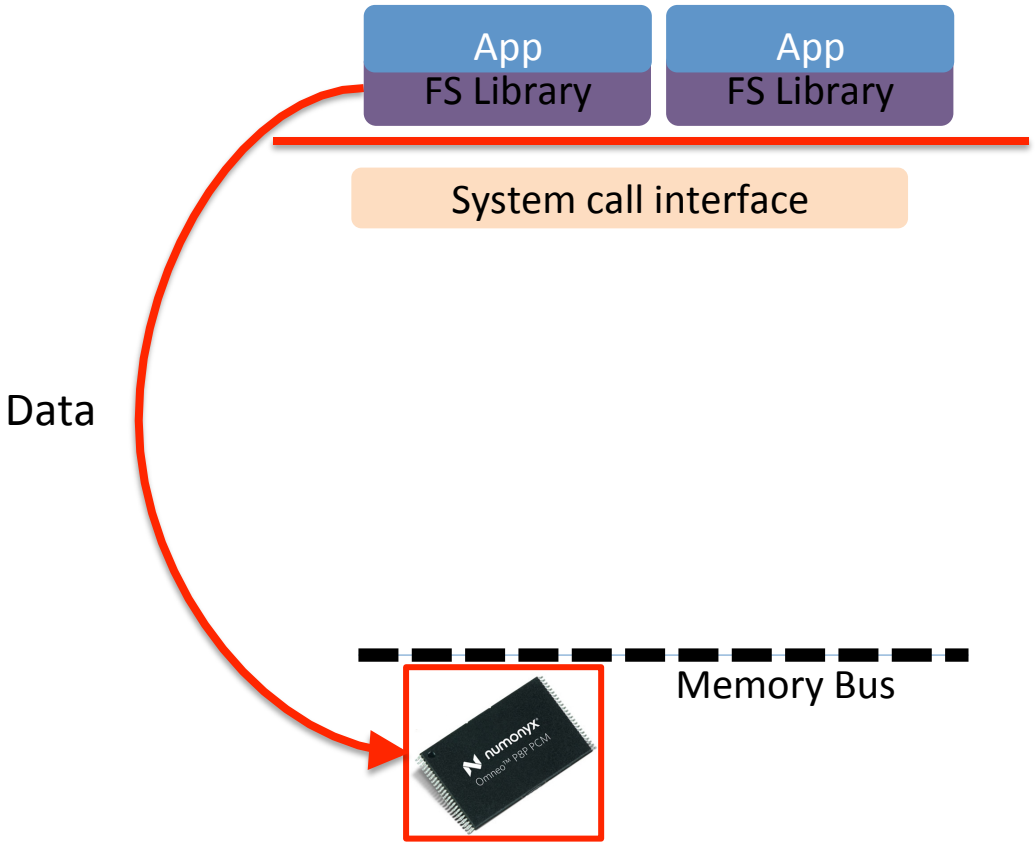
PMFS techniques

- Accidental write corruption
  - Leverage enable/disable write protection in kernel
  - Supervisor-mode Access Protection in user mode
- Log-based metadata transactions
  - Uses less space than copy-on-write
  - Avoids false sharing that copies large blocks
- Copy-on-write only for whole-block updates

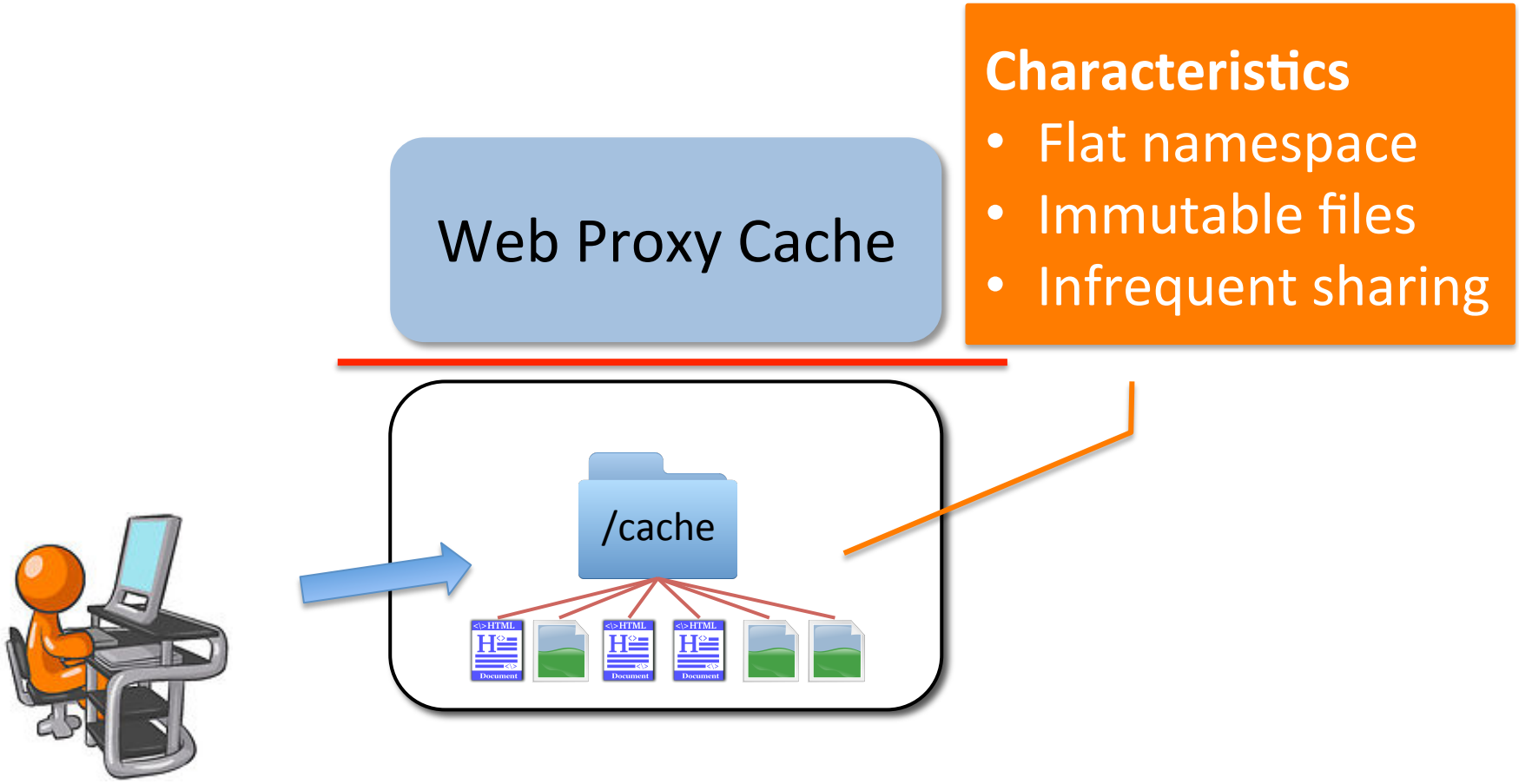
- Short-circuit common operations
  - In-place overwrite

User-Mode File Systems

- Benefit: latency, flexibility
- Challenge: synchronization, protection

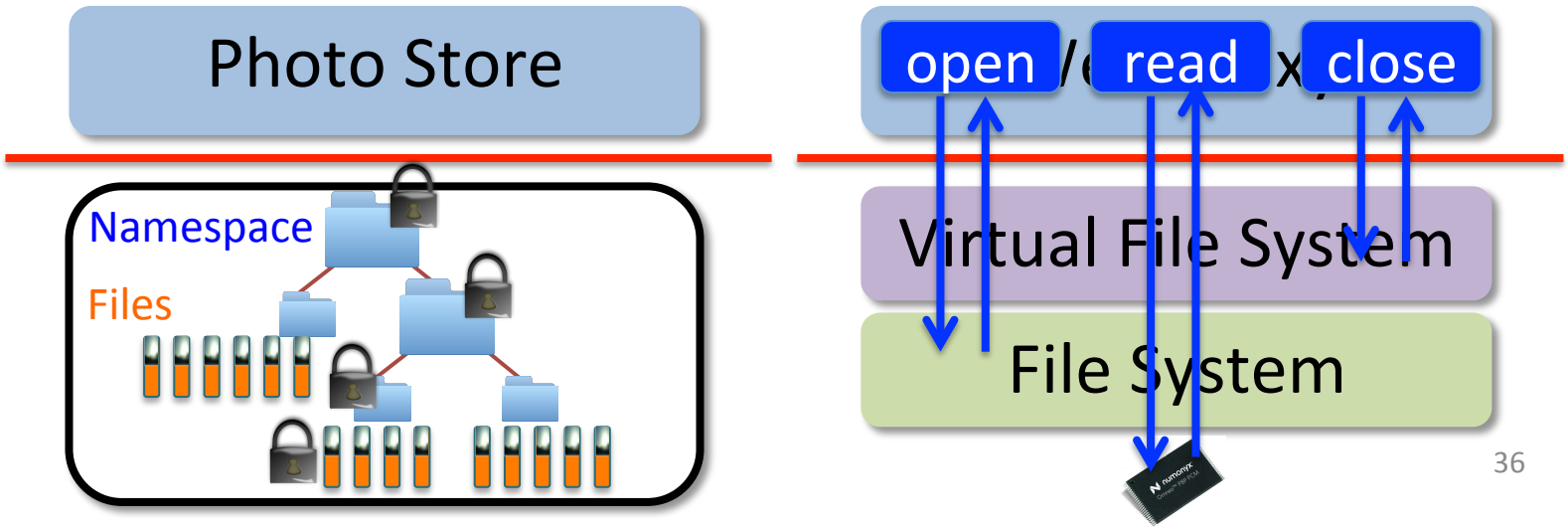


Example: Web proxy

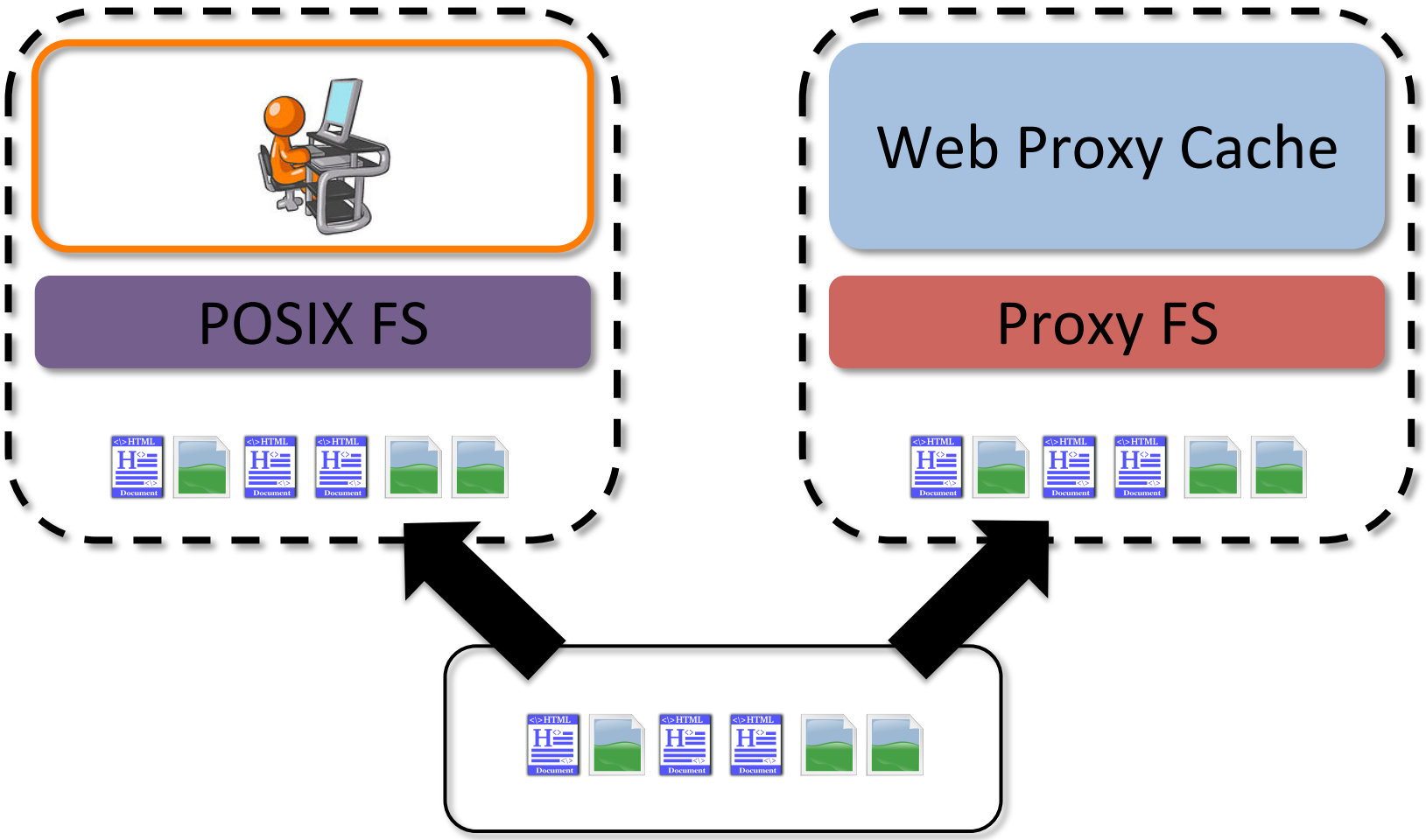


The Abstraction Cost of File

- Rigid interface and policies
  - Has fixed components and costs
  - Hinders application-specific customization

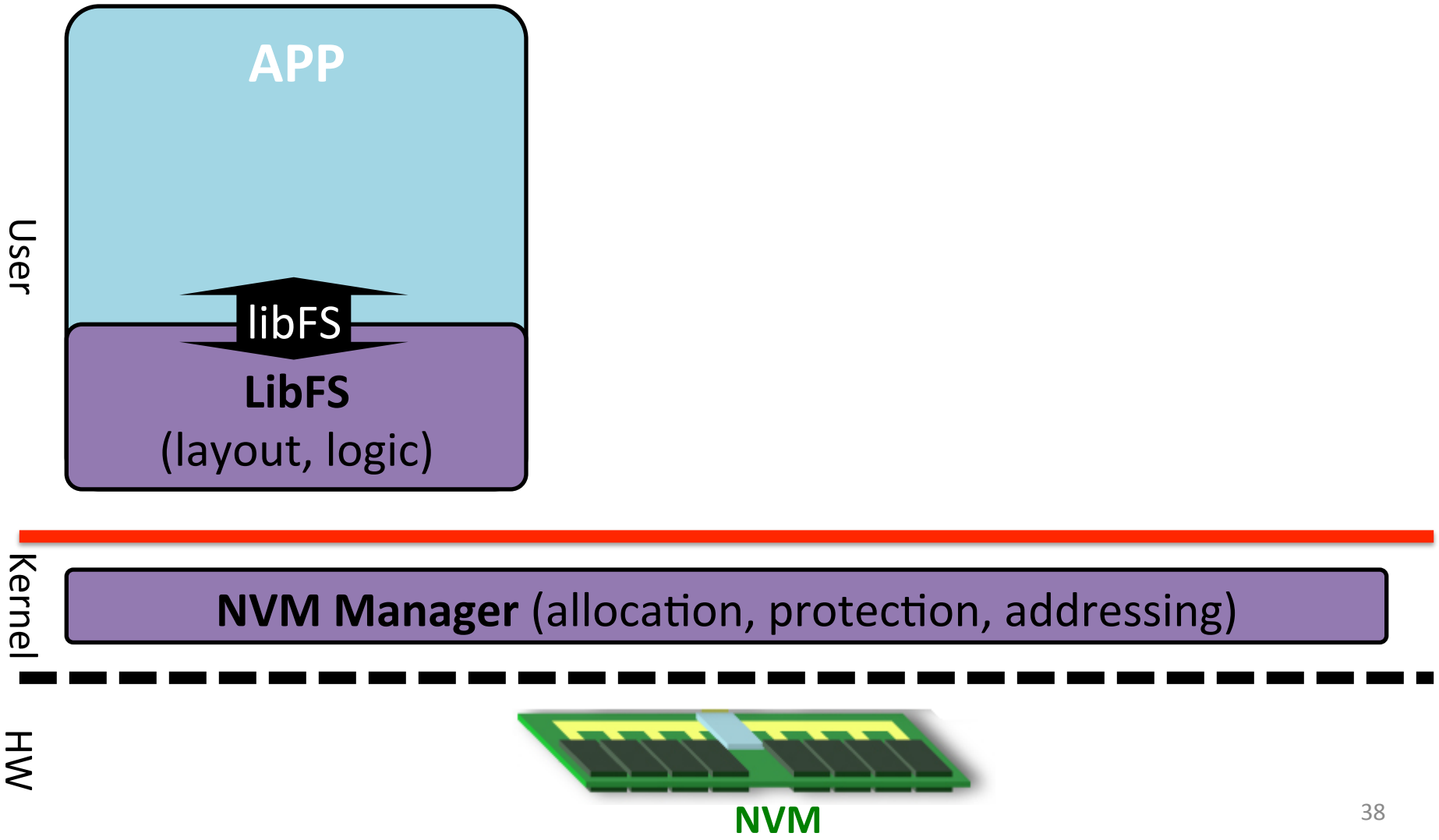


Aerie: Exokernel-like libFS

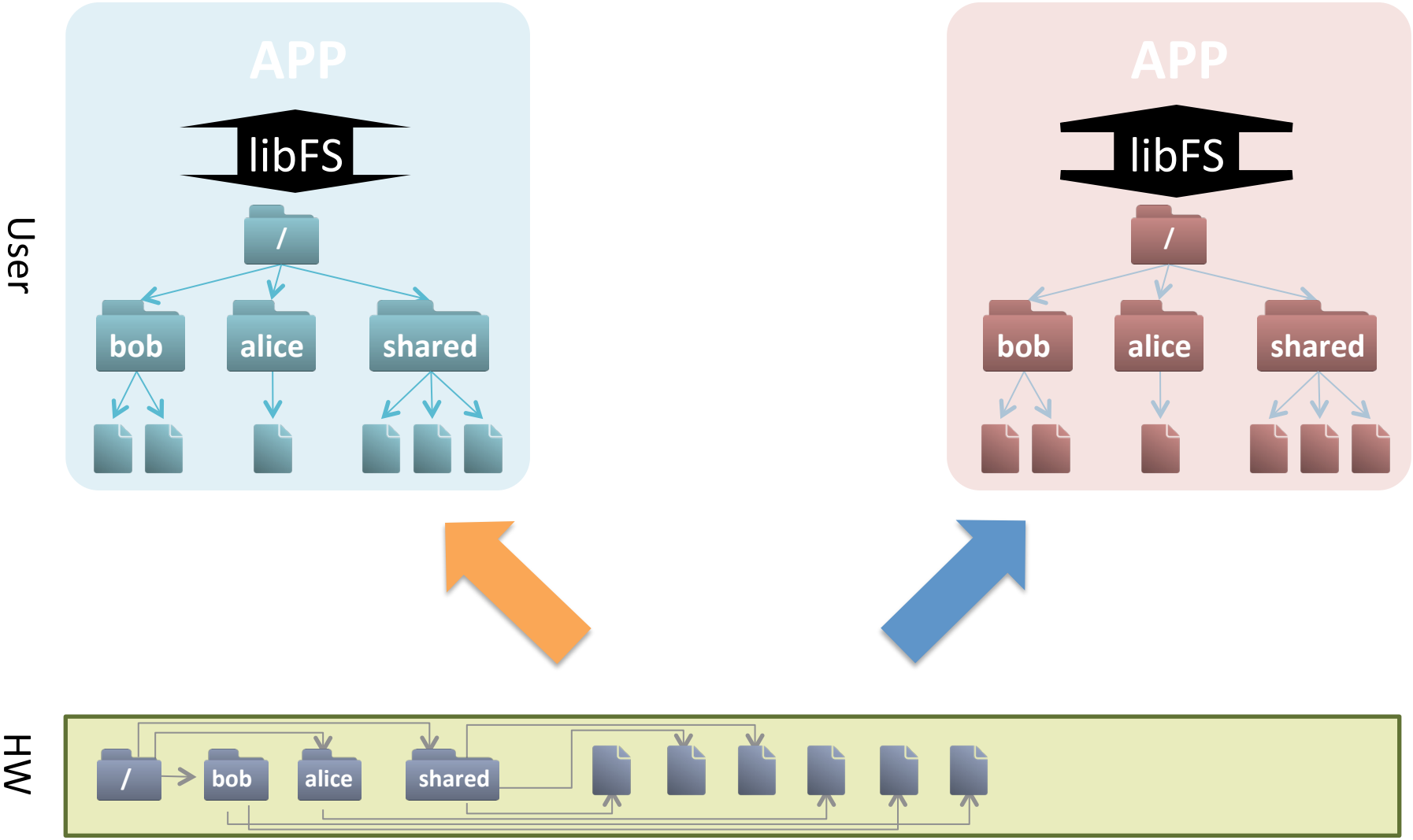


[Volos , EuroSys'14]

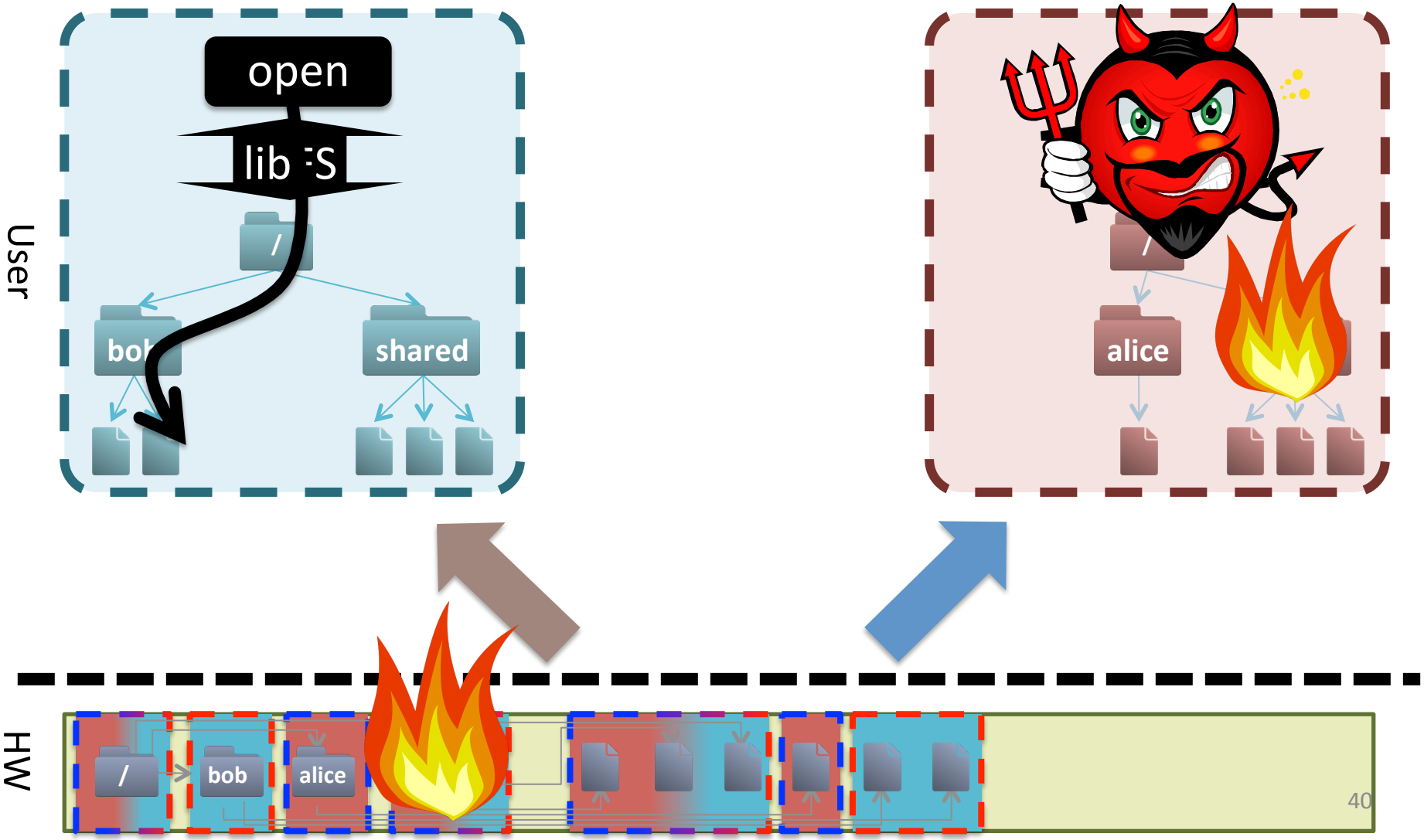
Aerie in a nutshell: Components



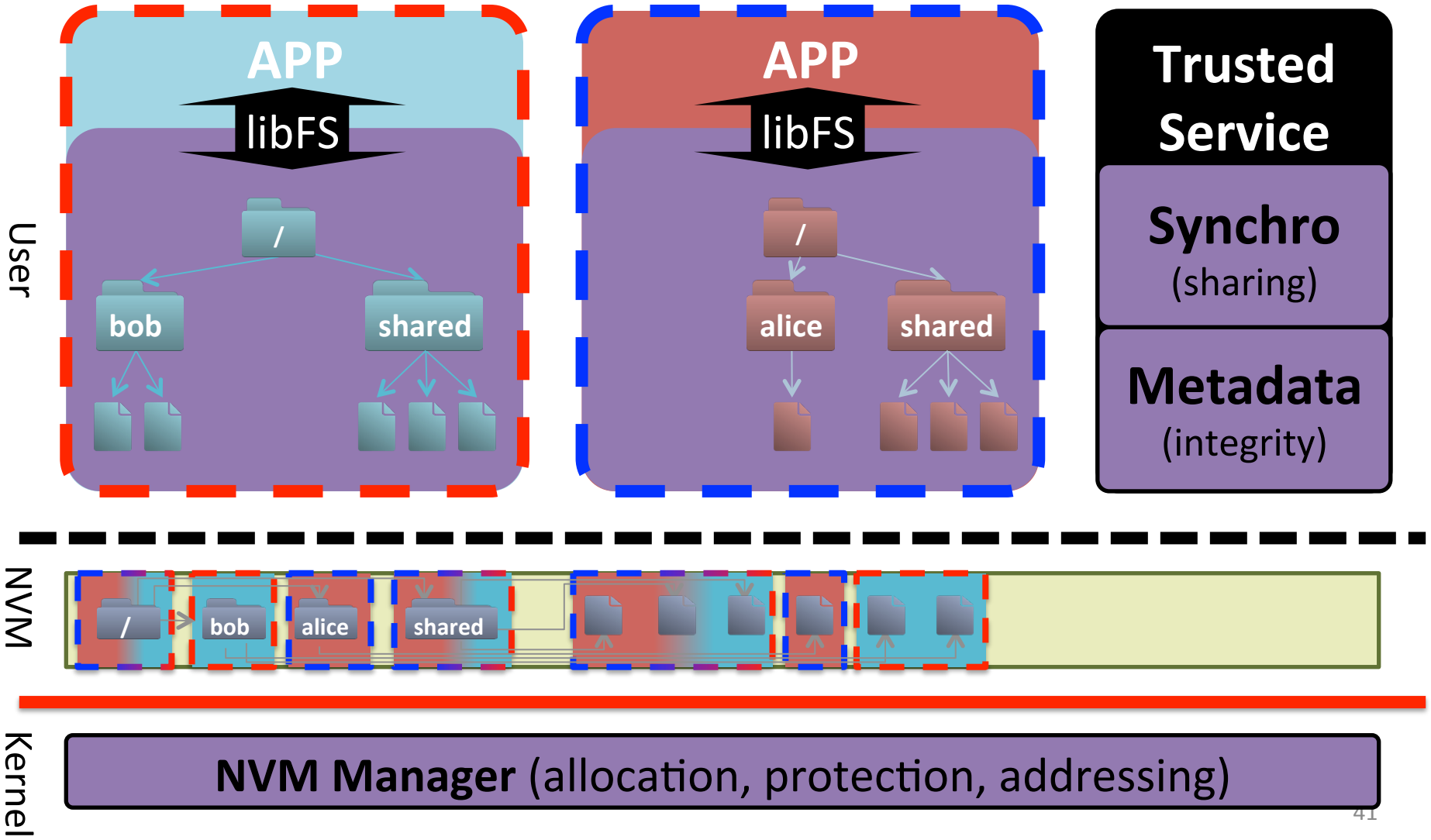
Aerie in a nutshell:
Operational view



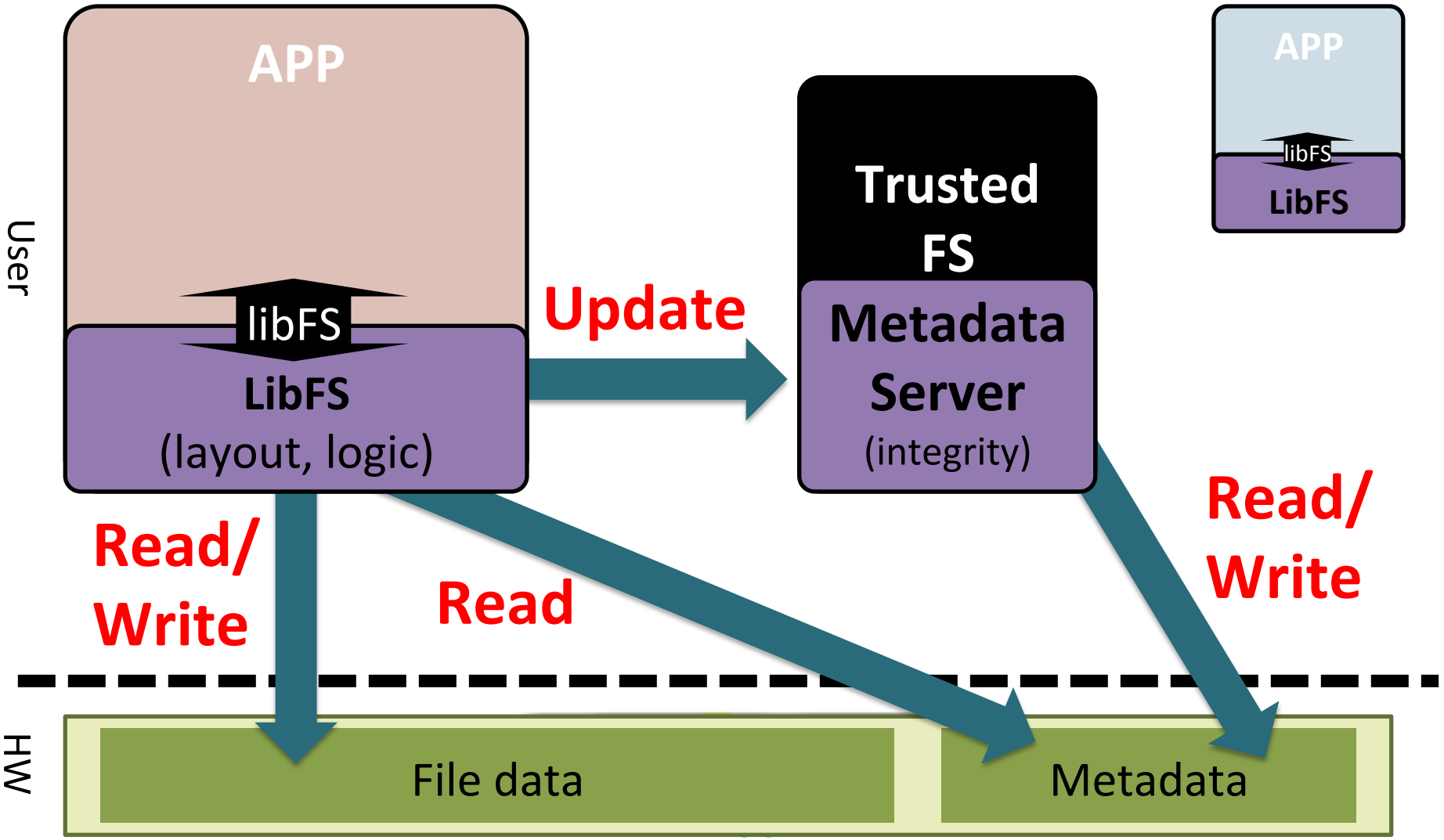
Aerie in a nutshell:
Operational view



Integrity and Synchronization

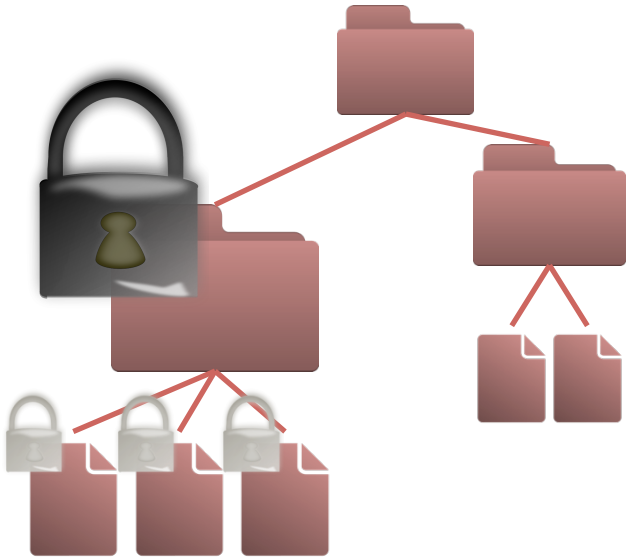


Aerie Access Pattern

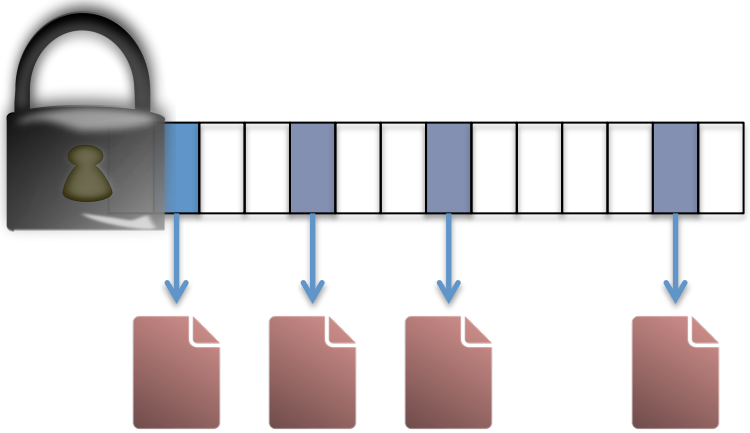


File system libraries

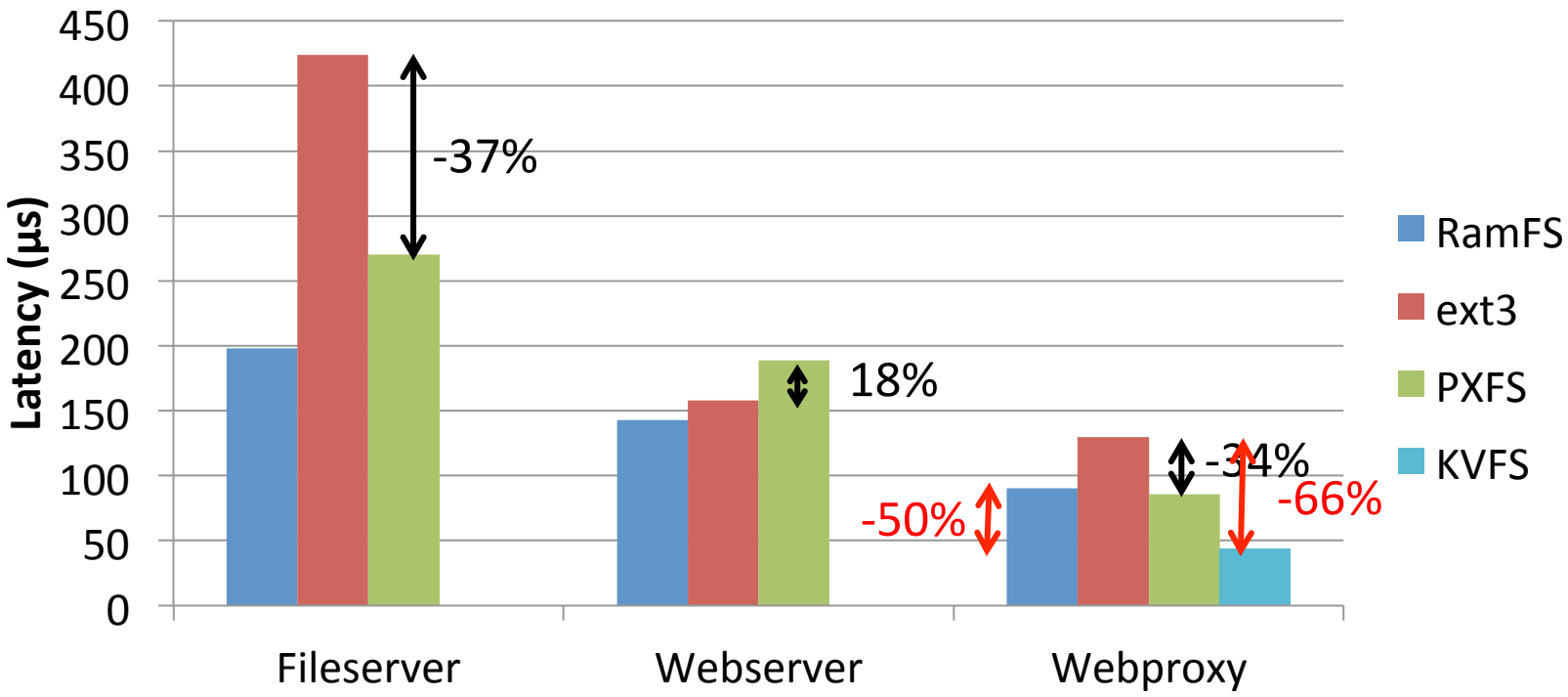
- PXFS: POSIX-style FS
  - POSIX interface
  - Hierarchical DAG namespace
  - Lock per file



- KVFS: Key-value FS
  - Put/Get/Erase
  - Flat key-based namespace
  - Global file-system lock

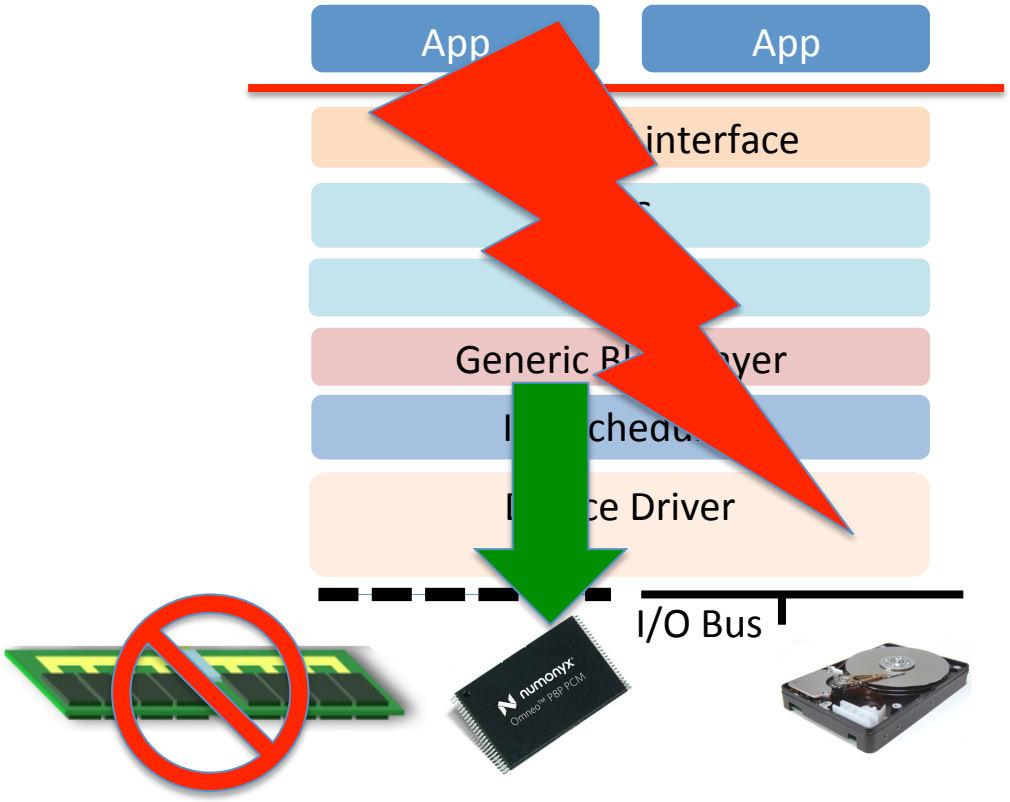


Application-workload performance



- PXFS performs as well or better than kernel-mode FS
- KVFS exploits app semantics to improve performance

Whole System Persistence



Whole System Persistence

- Use NVM exclusively
- On power failure, flush registers and cached data to NVM
- On restart, hot-plug devices and resume

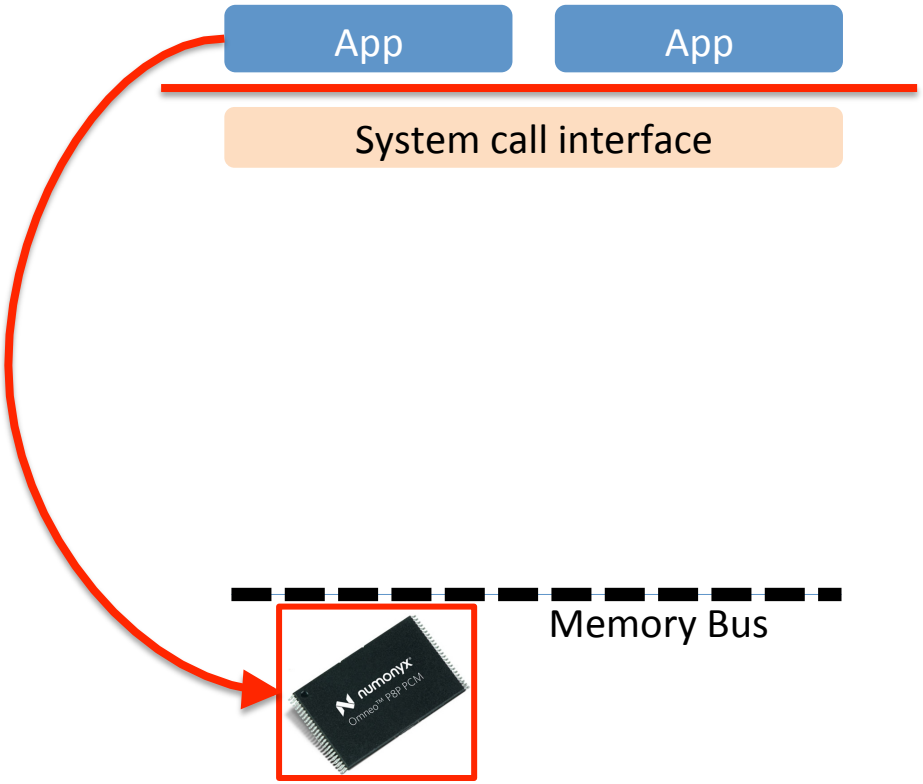
- Benefits:
  - No programming changes
  - fsync() = nop;

Outline

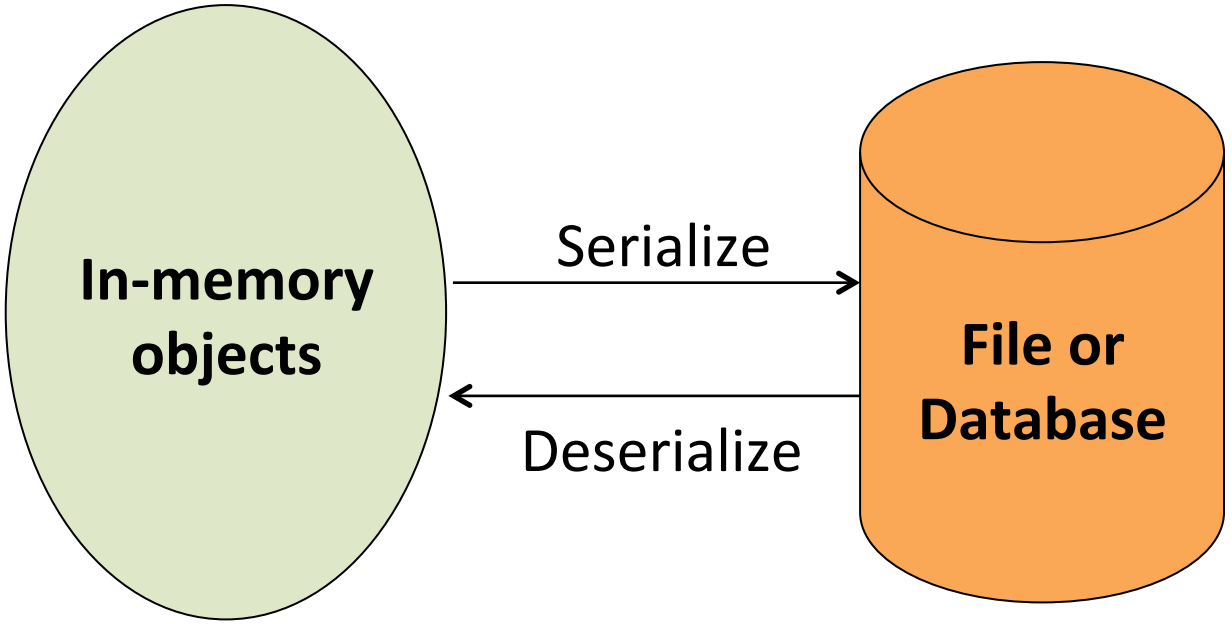
- NVM solid-state drives
- Persistent memory file systems
- Persistent regions/heaps
  - Mnemosyne, NV-Heaps
- Persistent data stores

User Persistent Memory

- Challenge: consistency

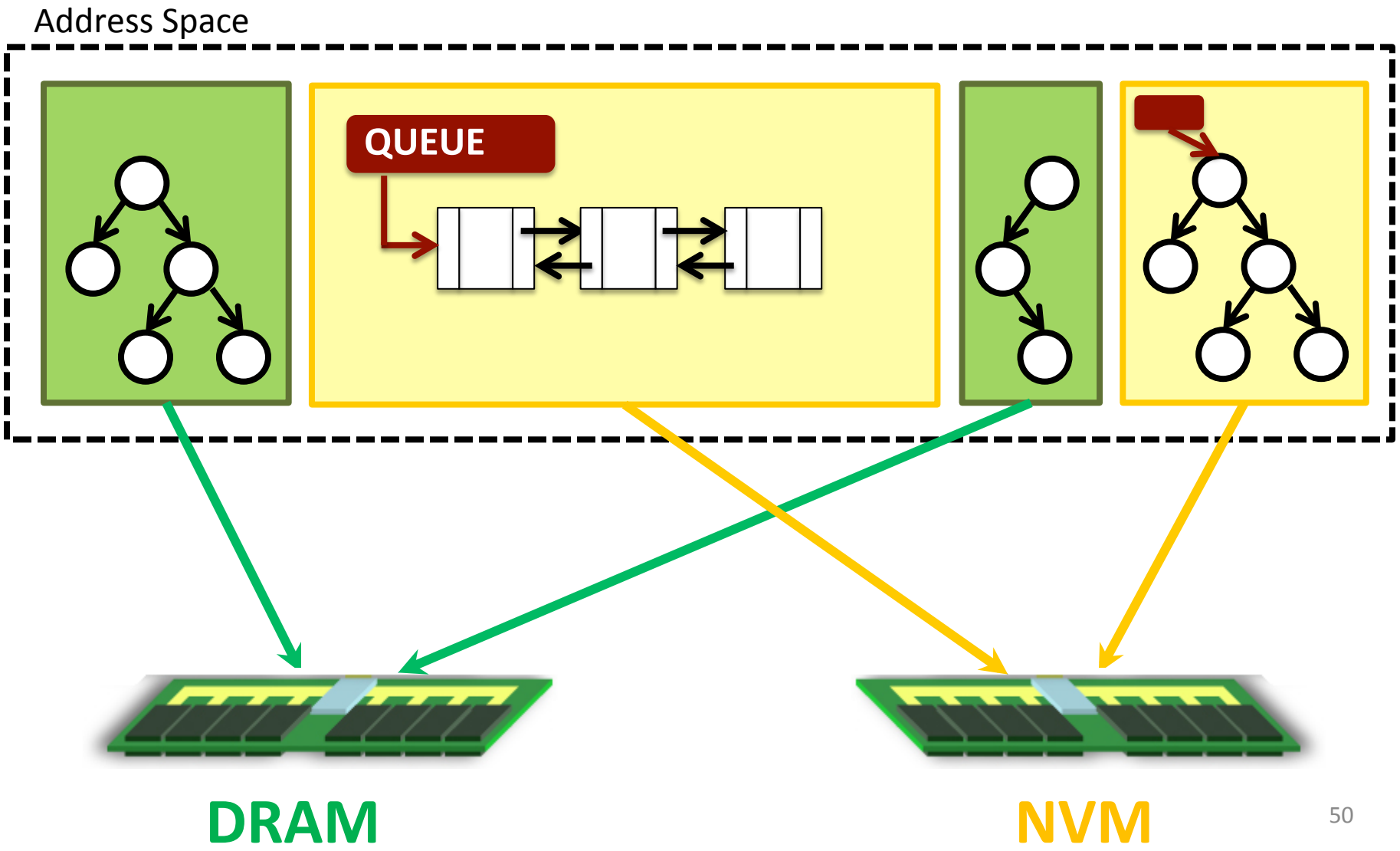


Traditional Approach to Durability

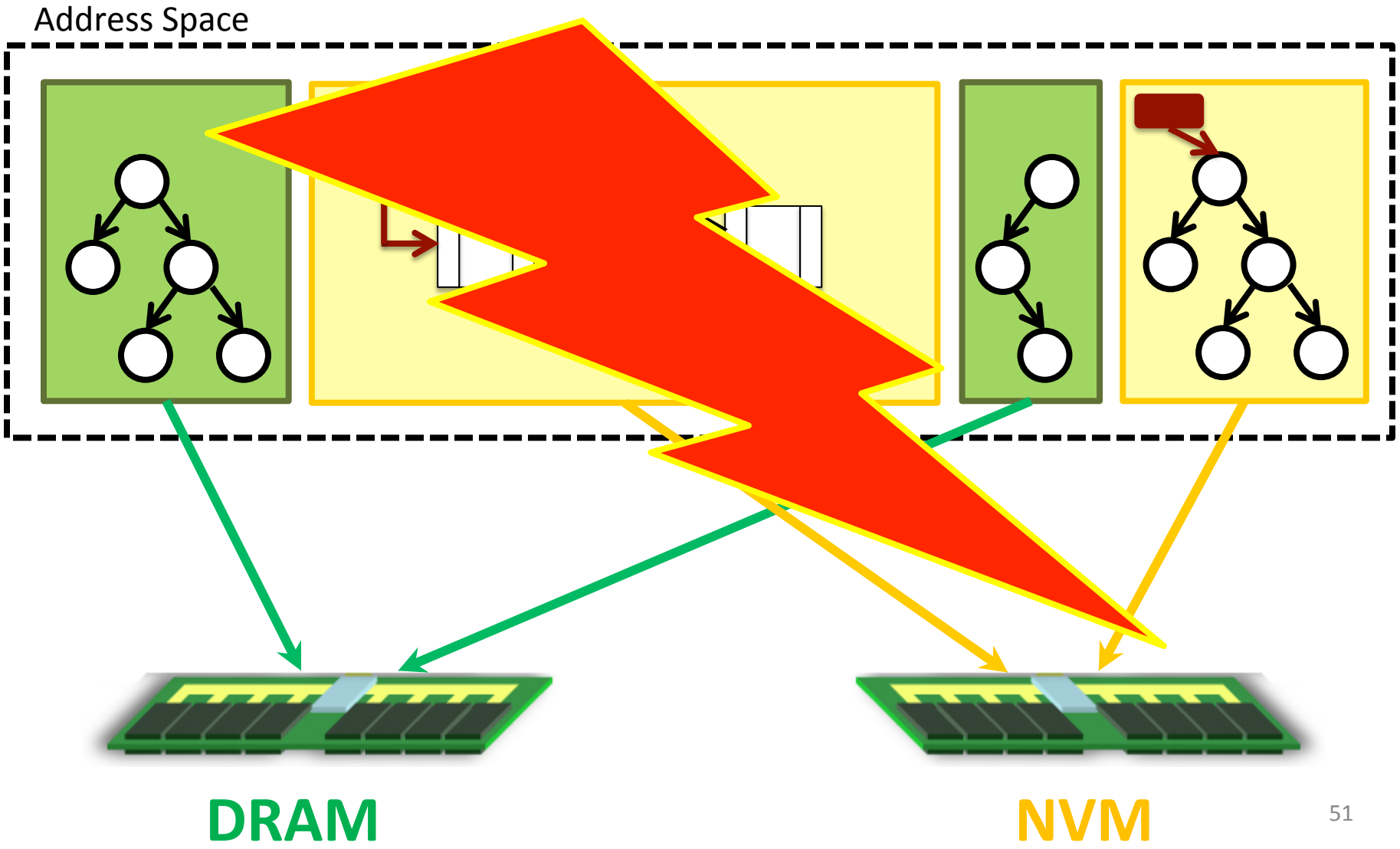


- Separate object and persistent formats
- Translation code
- Programmability and performance issues

Persistent Regions in a nutshell

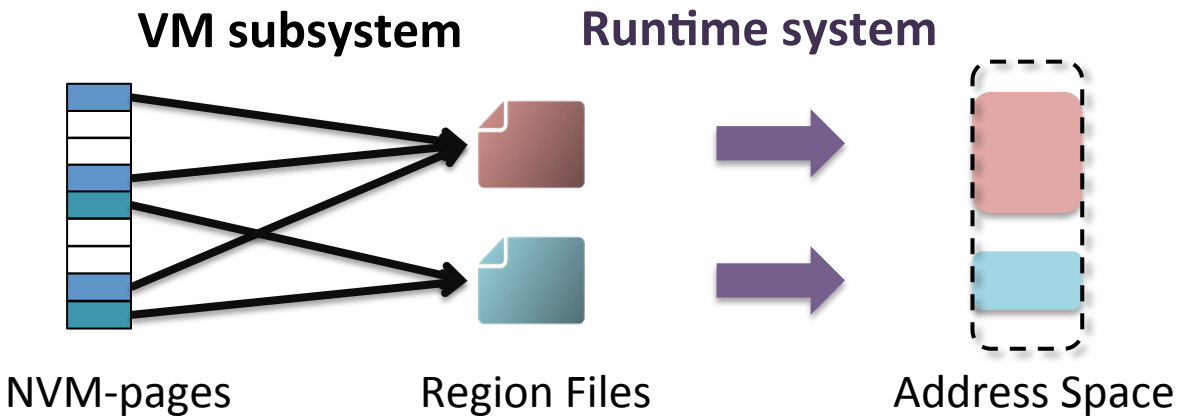


Persistent Regions in a nutshell



Persistent regions

- Virtual memory segments stored in NVM
  - Optionally virtualized to a region file on disk
- ➢ Implementation
  - Linux DAX: on mmap(), NVM pages mapped to user space



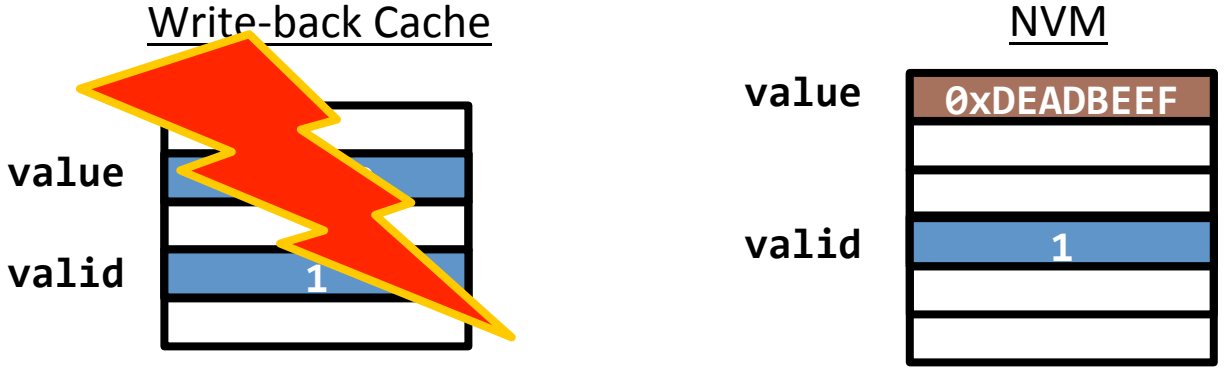
Consistent updates

- Rely on hardware for *1-word* atomic update

```
STORE value = 0xC02
STORE valid = 1
```

➢ CPU cache may **reorder** writes to NVM

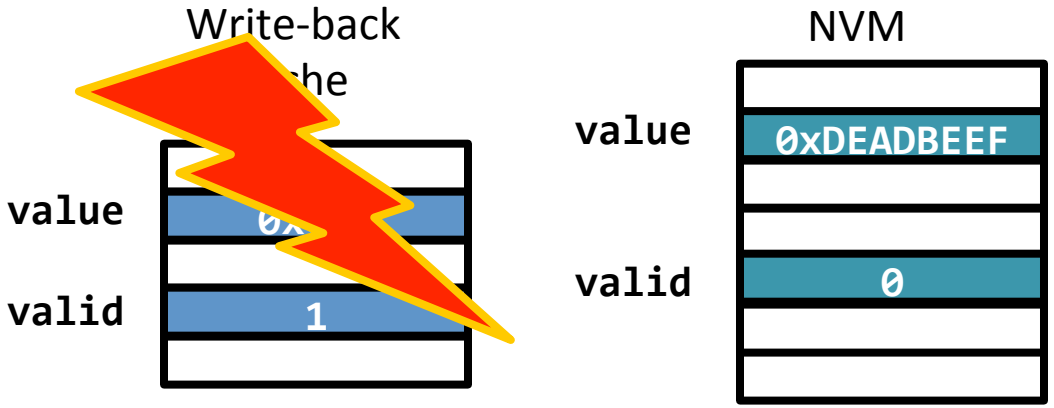
– Breaks *"crash-consistent"* update protocols



Primitive operation: ordering writes

- Why?
  - Ensures ability to **commit** a change
- How?
  - Flush – **MOVNTQ/CLFLUSH**
  - Fence – **MFENCE**
- Inefficiencies:
  - Removes recent data from cache

```
STORE value = 0xC02
FLUSH (&value)
FENCE
STORE valid = 1
```

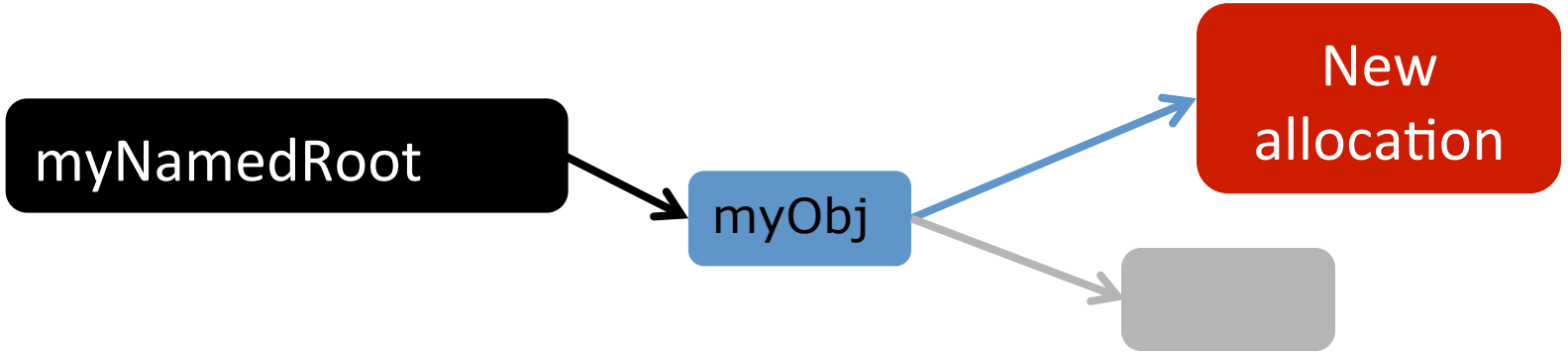


Allocating Persistent Memory

- pstatic var
  - Allocates a persistent variable in a static region
- pmalloc(sz, addr)
  - Allocates a persistent memory chunk

Avoiding Persistent Memory Leaks

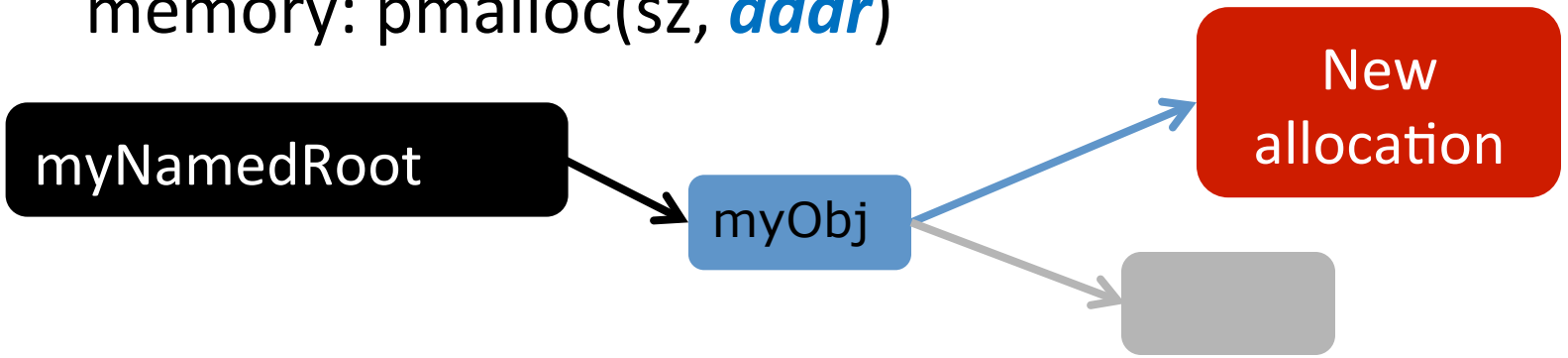
- Programmer shall ensure reachability



pstatic void * myNamedRoot;

Avoiding Persistent Memory Leaks

- Programmer shall ensure reachability
- Mnemosyne API helps programmer
  - Atomically sets pointer to newly allocated memory: pmalloc(sz, ***addr***)



pstatic void * myNamedRoot;
pmalloc(…, *&myNamedRoot*);
pmalloc(…, *&(myObj->child)*);

[Volos, ASPLOS'11]

Avoiding Persistent Memory Leaks

- Programmer shall ensure reachability
- Mnemosyne API helps programmer
  - Atomically sets pointer to newly allocated memory: pmalloc(sz, ***addr***)
  - Raises compiler-time warnings when target types mismatch:

```
void persistent * pptr;   // points to persistent data
void * vptr;   // points to volatile data
…
vptr = pptr;
```

!!! WARNING

Consistent Updates

- Support updating data without risking correctness after a failure

➢ Four update operations

Update	API	Usage example
Single-variable	HW primitive	Set flag
Append	`log_append`	Append to journal
Shadow	`pmalloc` + single-var	Update tree node
In-place	`patomic { }`	Update double linked list

Single-variable Updates

- Example: Set a flag

```
// update data
…
node->valid_data = 1;
```

Single-variable Updates

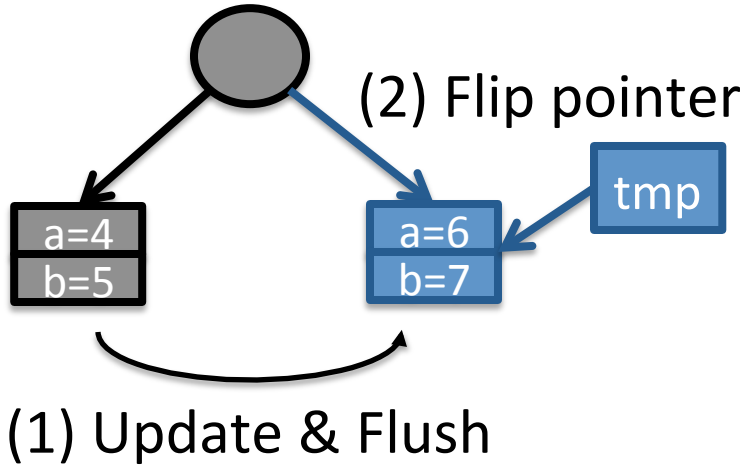
- Example: Set a flag

```
// update data
…
store(&node->valid_data, 1);
flush(&node->valid_data); // flush cache line
fence(); // wait for cache-line flush to complete
```

Limitation: Single-word atomicity

Shadow Update

- Example: Update a tree node



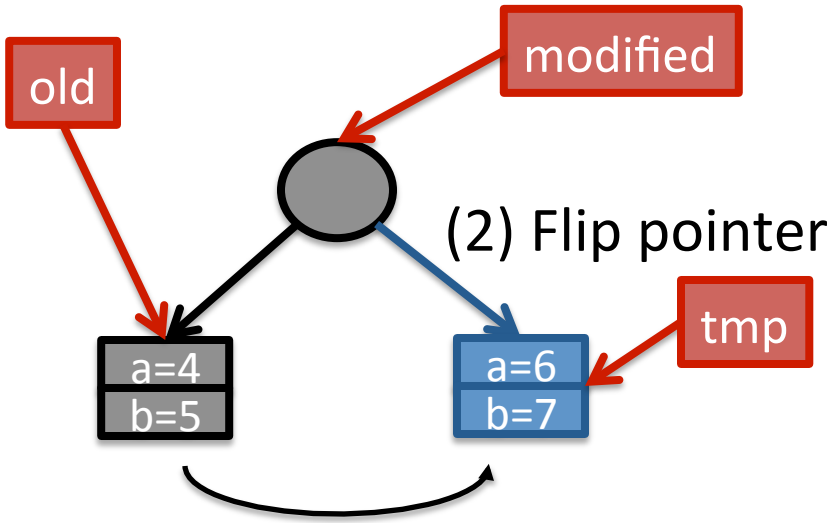

```
pmalloc(…, &tmp);
tmp->a = 6;
tmp->b = 7;
flush(&tmp);
fence();
parent->child = tmp;
```

Not exactly right: we can still lose space

Works better with custom allocator

Shadow Update

- Example: Update a tree node




```
old = parent->child;
modified = parent
flush(&old);
flush(&modified);
fence();
pmalloc(…, &tmp);
tmp->a = 6;
tmp->b = 7;
flush(&tmp);
fence();
parent->child = tmp;
…
```

Intentions

Log-based Journaling

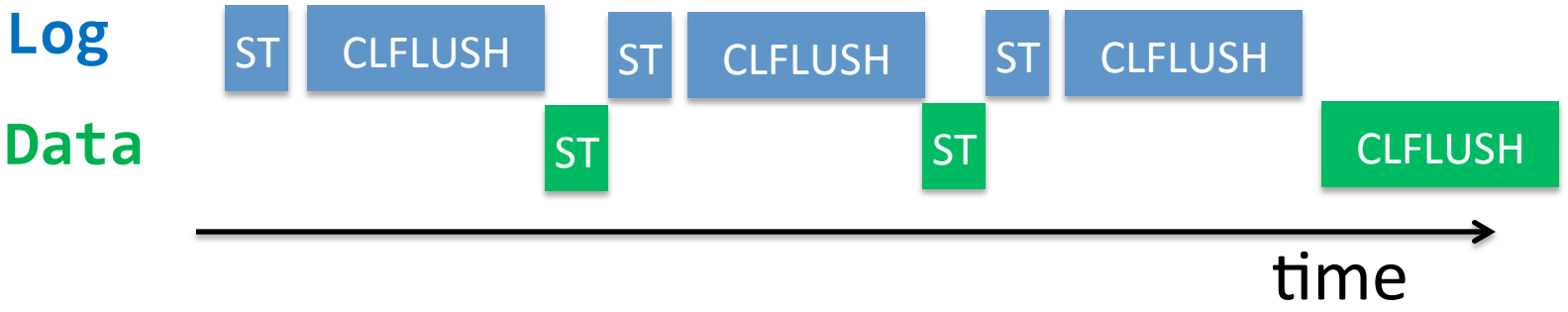
- Log intentions

```
log_append(NEW_NODE);
log_append(parent);
log_append(old_node);
log_append(new_node);
log_flush(); // make log stable
// update node
log_truncate()
```

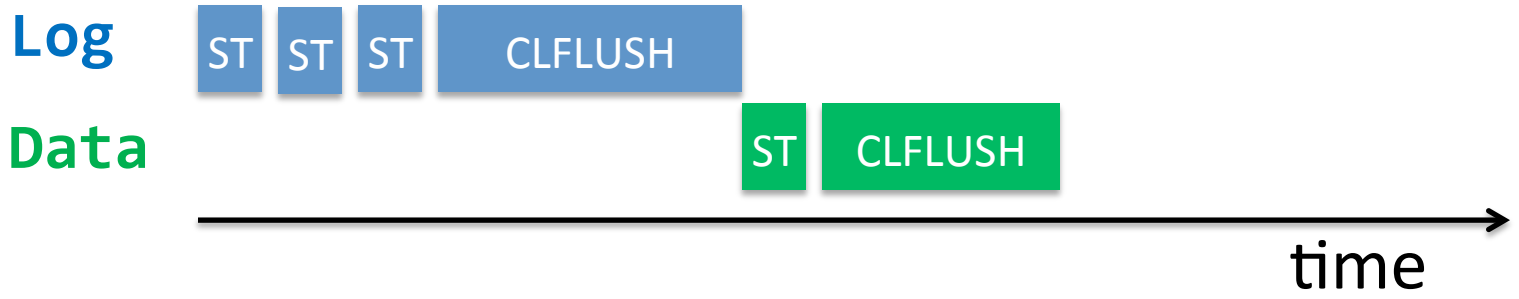
- Inspect log upon recovery and undo incomplete actions

Undo vs Redo Logging

- Write old data durably to log before in-place update



- Write new data durably to log before in-place update



Consistent Durable Data Structures

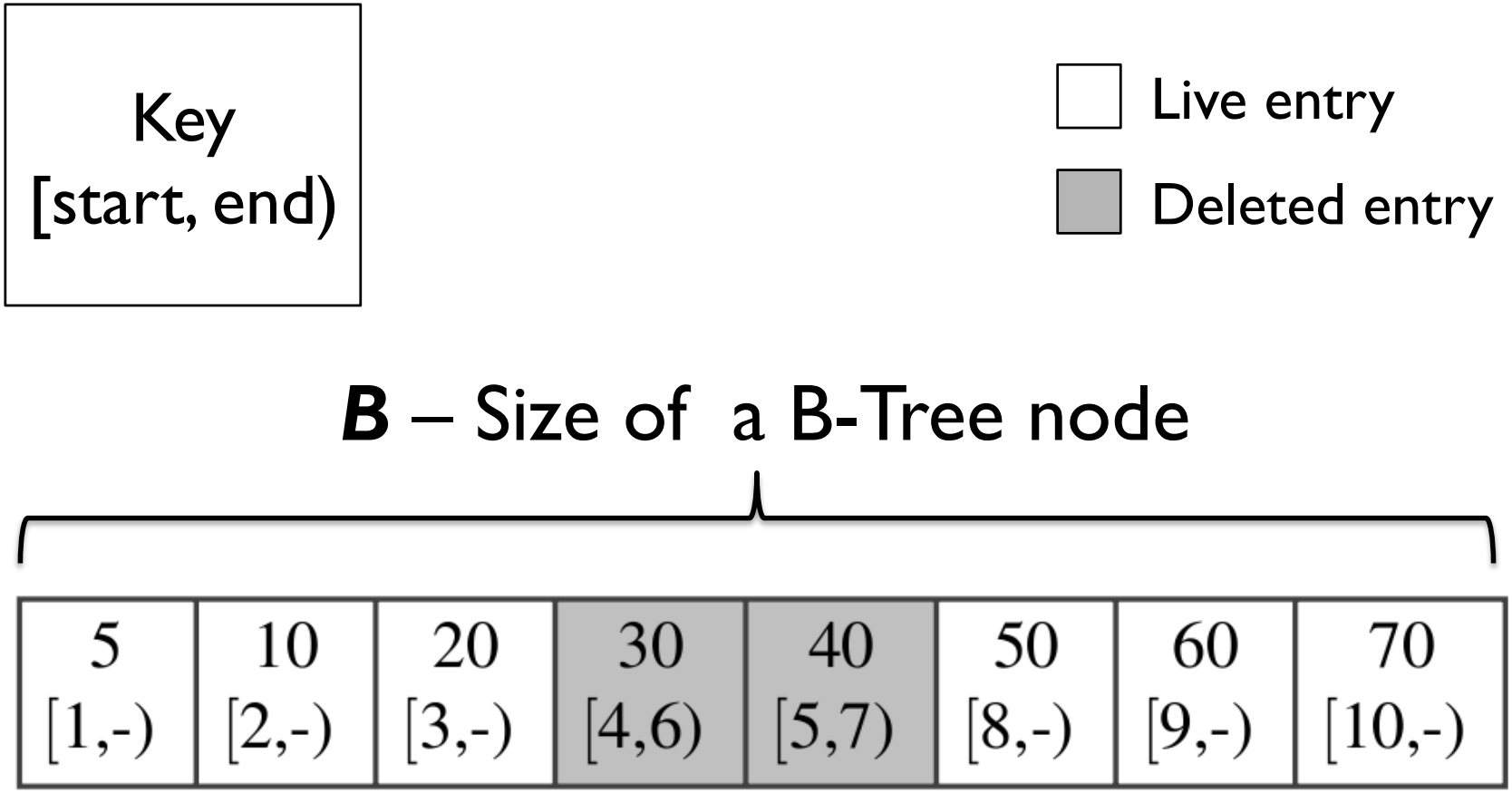
- Every update creates a new version
- Data representing older versions are never overwritten during an update
  - Modified atomically or copy-on-write.
  - Flushed/fence after all updates written
- Last consistent version
  - Stored in a well-known location
  - Fence before update
  - Used by reader threads and for recovery

[Venkataraman, FAST'11]

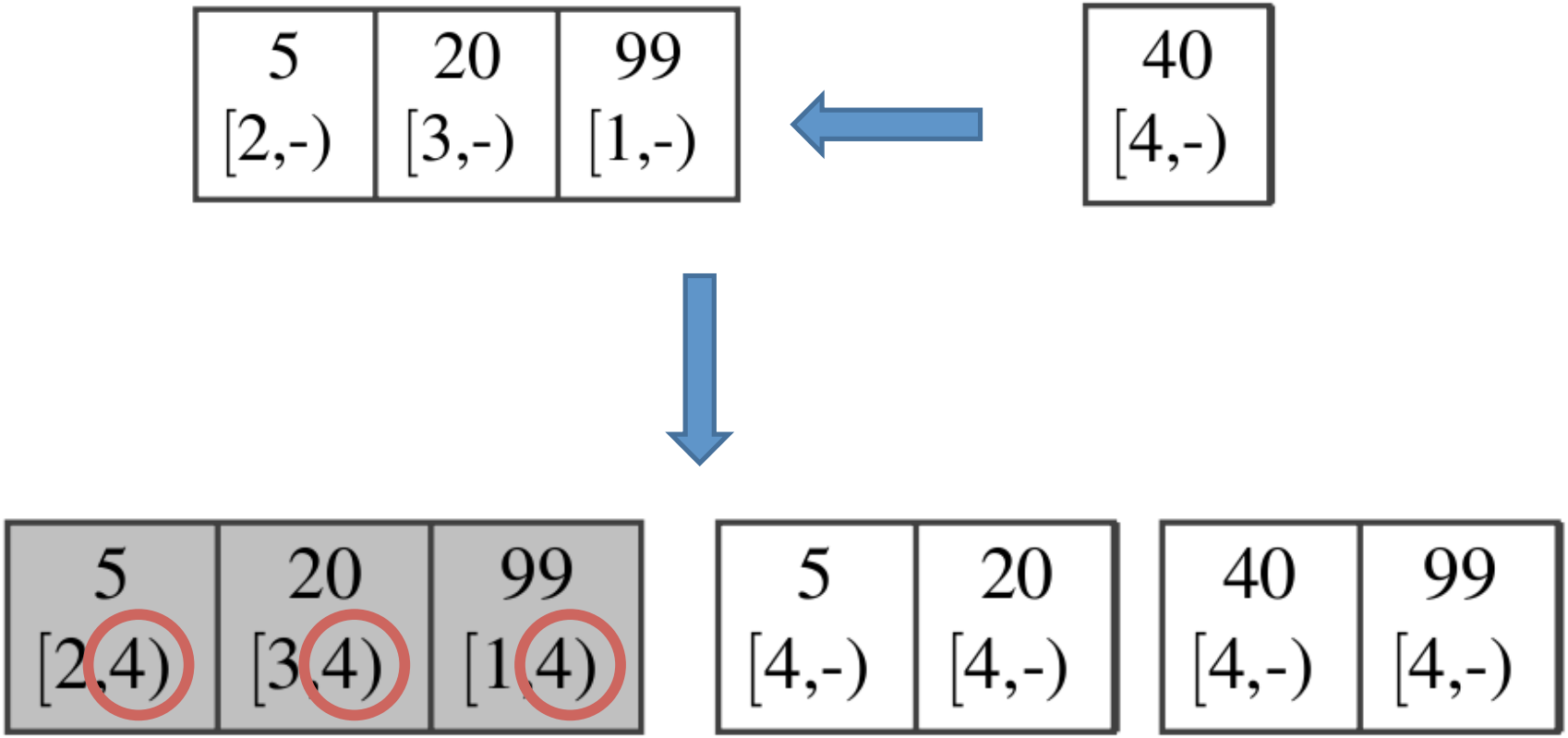
Garbage Collection

- CDDS tracks oldest version with non-zero references
- Collect data with older version numbers

CDDS B-Tree Node

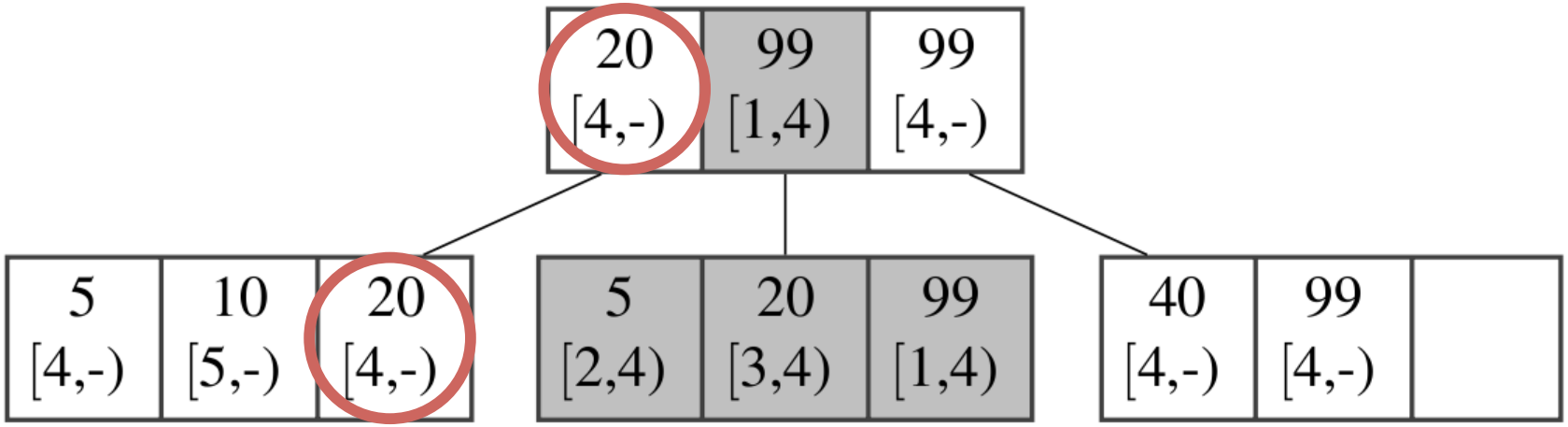


B-Tree Insert/Split



CDDS B-Tree Lookup

Find key 20 at version 5



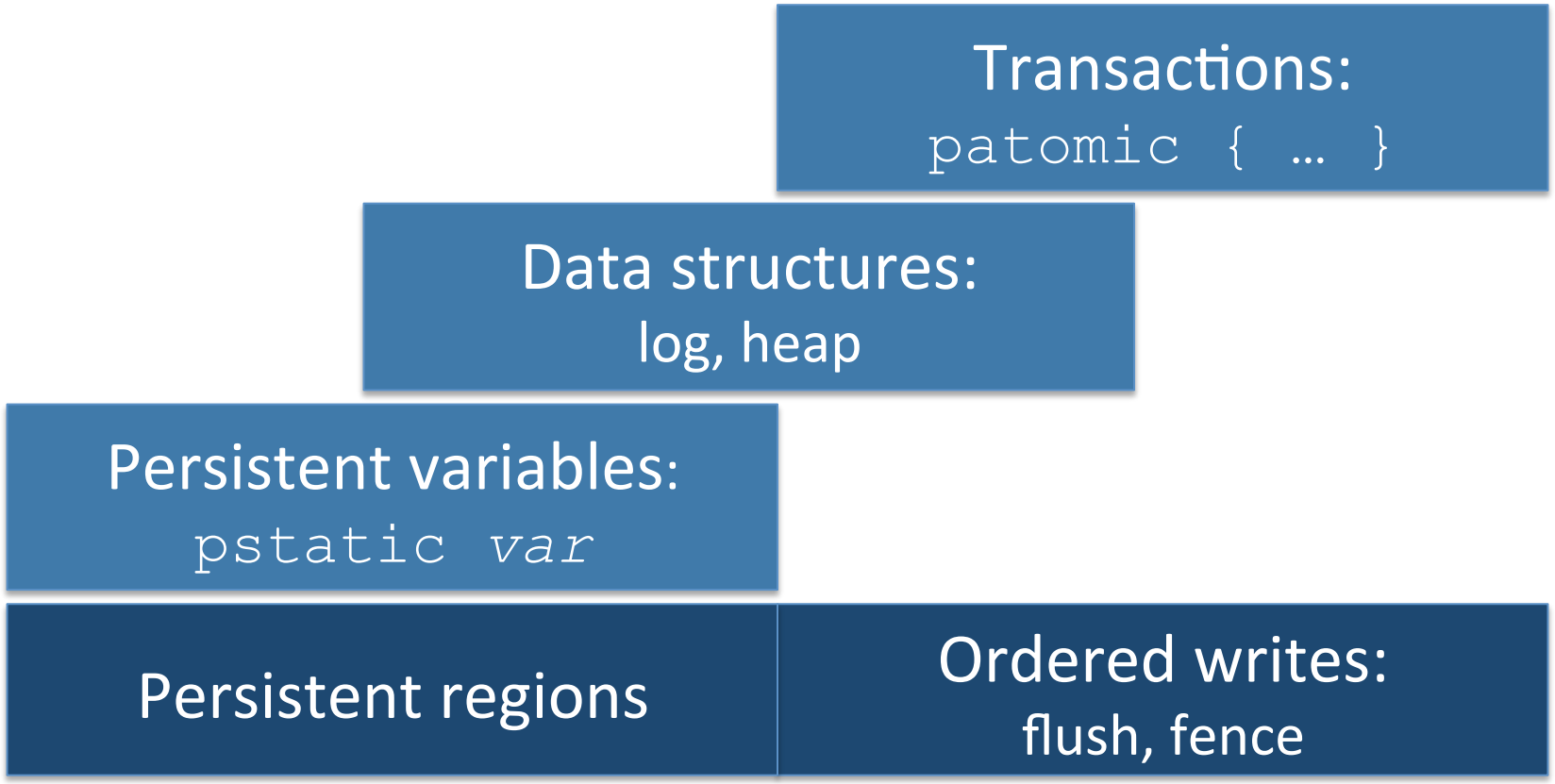
Recovery

- On program restart, programs restart as normal; not from a checkpoing
  - Integrate persistent memory,
  - Replay logs, garbage-collect data structures

Software Wear-Leveling

- Most NVMs wear out after 10^8-10^{15} overwrites
- Allocators can reduce overwrite
  - Allocate new space instead of re-using old blocks
- Shadow paging reduces overwrite
  - New data written to new locations
  - Hot data is spread out over many places

Programming Persistence Levels



Durable memory transactions


```
patomic {
    B.next = C;
    C.prev = B;
}
```

Durable memory transactions


```
C.val = X;
patomic {
    B.next = C;
    C.prev = B;
}
```

Completion of patomic does NOT guarantee persistence of C.val

Durable memory transactions


```
C.val = X;
flush(&C.val);
fence();
patomic {
   B.next = C;
   C.prev = B;
}
```

OR

```
patomic {
   C.val = X;
   B.next = C;
   C.prev = B;
}
```

Durable memory transactions

- **Intel STM Compiler** instruments atomic blocks

```
patomic {
 B.next = C;
 C.prev = B;
}
```

```
begin_transaction();
stm_store(&B.next, C);
stm_store(&C.prev, B);
commit_transaction();
```

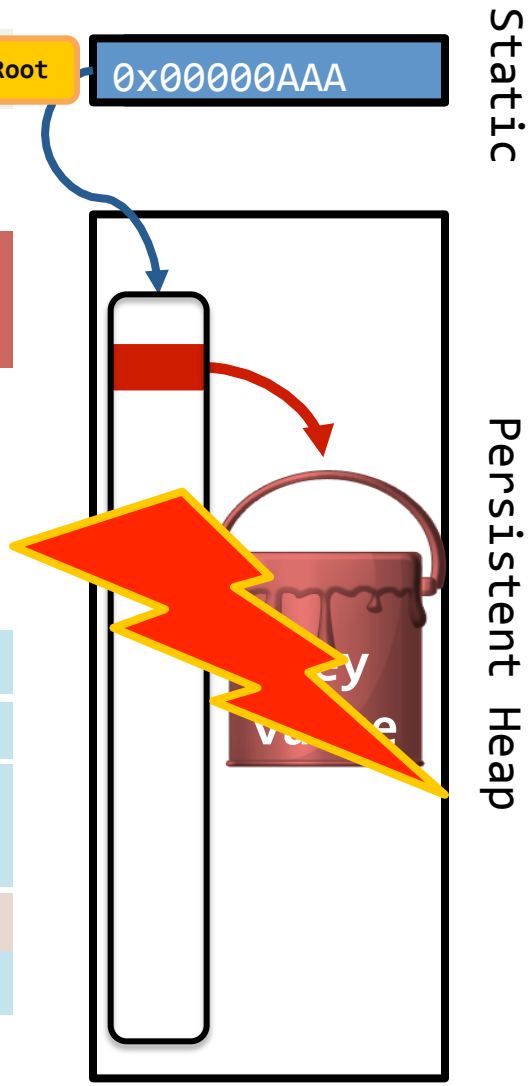
- **libmtm** runtime supports ACID transactions
  - Log-based recovery after crash

Hash table example

```
pstatic htRoot = NULL;

main() {
  if (!htRoot)
    pmalloc(N*sizeof(*bucket), &htRoot);
}


update_hash(key, value) {
  patomic {
    pmalloc(sizeof(*bucket), &bucket));
    bucket->key = key;
    bucket->value = value;
    insert(hash, bucket);
  }
}
```



NV-Heaps

- NV-Heaps programming interface supports
  - Persistent objects as the underlying data model
  - Referential integrity via specialized pointer types
  - Crash-consistent updates via transactions

- NV-Heaps exist as self-contained relocatable memory-mapped files

[Colburn, ASPLOS'11]

Smart Pointer Types

- Prevent unsafe references
  - NO non-volatile pointers to volatile data
  - NO inter-heap pointers
- Prevent persistent memory leaks
  - Pointers maintain per-object reference count
  - Objects have to be reachable from the heap's root
- Support NV-Heaps relocation
  - Hold offset rather than absolute address

NV-Heap API Example

```
class NVList: public NVObject {
  DECLARE_POINTER_TYPES(NVList);
public:
  DECLARE_MEMBER(int, value);
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
}
```

Make this a persistent object

Declare smart pointer types

Declare transactional accessors (typical in Object-based STM)

```
NVHeap* nv = NVHOpen("foo.nvheap");
NVList::VPtr a = nv->GetRoot<NVList::NVPtr>();
AtomicBegin {
 ... = a->get_next();
 a->set_next(...);
} AtomicEnd
```

Open a heap file

Get heap's root

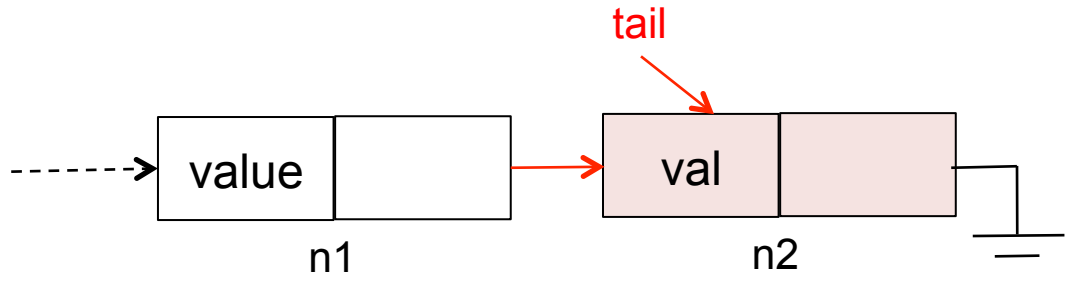
Begin a transaction

Durability Semantics for Lock-based Code

[OOPSLA14]

- Treat unlocked program points as consistent
- Leverage thread-consistent points to identify failure-atomic sections
- Example: adding to a shared queue

```
enqueue(val) {
    allocate_node()
    init_node()
    ---------------------
    L.lock()
    attach_node()
    move_tail()
    L.unlock()
    ---------------------
}
```



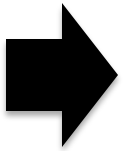
[Chakrabarti, OOPSLA'

Durability Semantics for Lock-based Code

[OOPSLA14]

- Treat unlocked program points as consistent
- Leverage thread-consistent points to identify failure-atomic sections
- Example: adding to a shared queue

```
enqueue(val) {
    allocate_node()
    init_node()
    L.lock()
    attach_node()
    move_tail()
    L.unlock()
}
```



```
enqueue(val) {
    pmalloc()
    init_node()
    L.lock()
    attach_node()
    move_tail()
    L.unlock()
}
```

happens before ⇒ durable

failure-atomic section ⇒ durable

Consistent Updates – Summary

- Support updating data without risking correctness after a failure

➢ Low-level mechanisms: Powerful for the expert

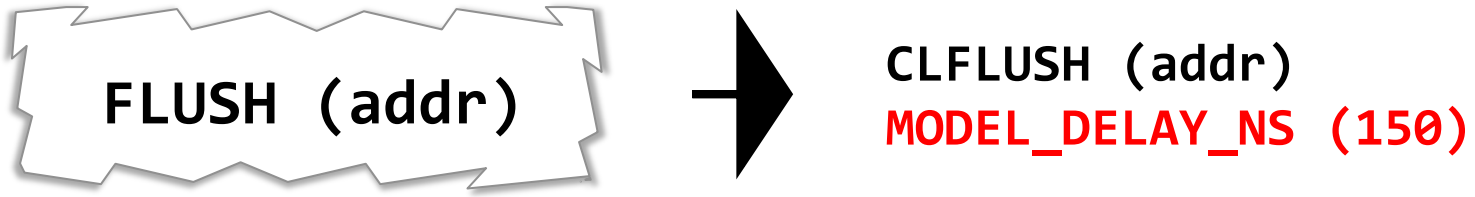
Update	API	Usage example
Single-variable	HW primitive	Set flag
Append	`log_append`	Append to journal
Shadow	`pmalloc` + single-var	Update tree node

➢ Transactions: Useful for common users

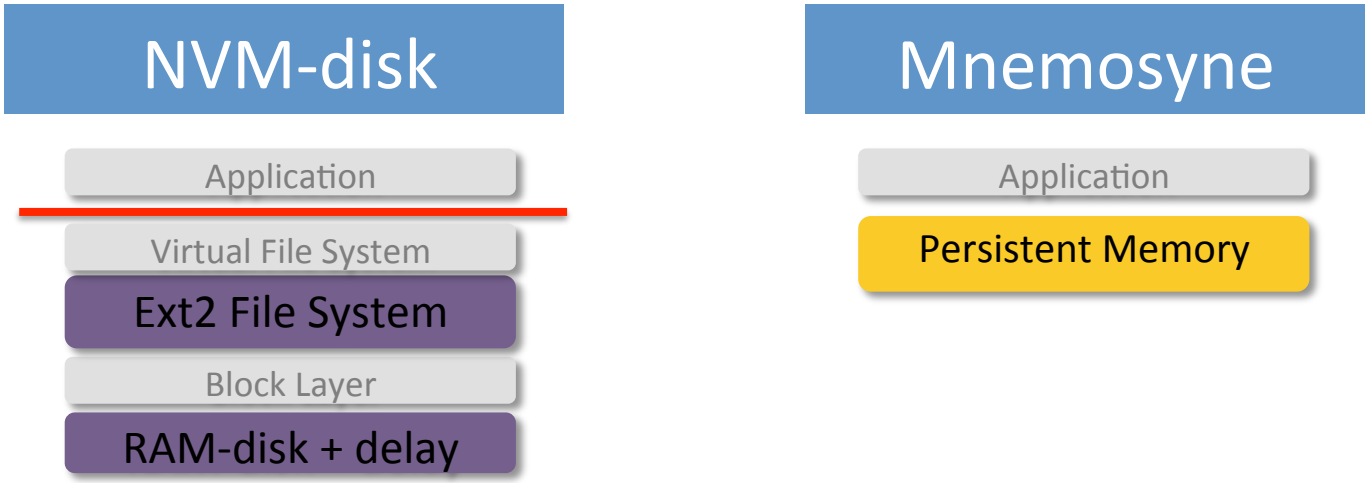
In-place	**`patomic { }`**	**Update double linked list**

Performance Evaluation

- NVM performance model based on DRAM



- Configurations

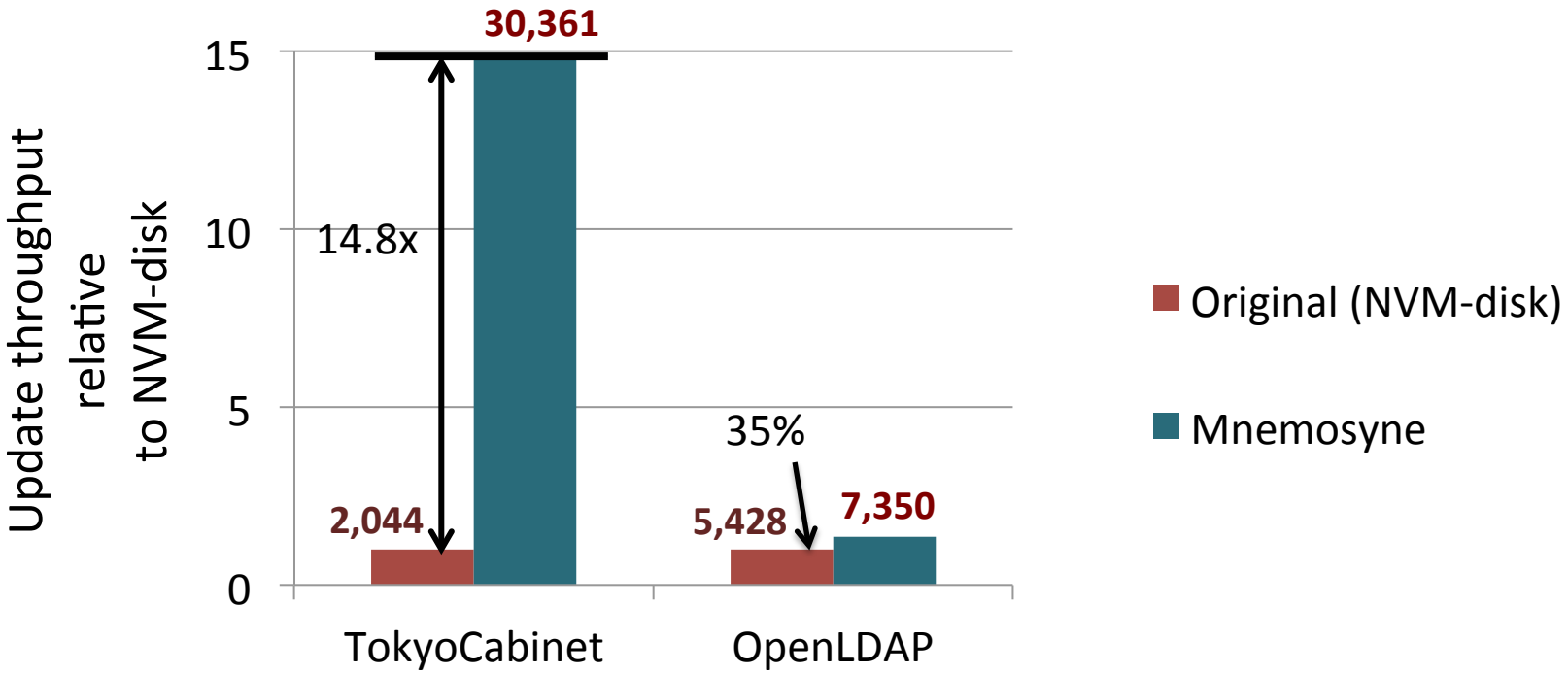


- Platform: Intel Core 2 2.5GHz (4 cores)

Applications

- TokyoCabinet: Key-value store
  - Original: msyncs B-tree to a mmap'd file
  - Modified: keeps B-tree in persistent memory
- OpenLDAP: Directory service
  - Original: stores dir-entries in Berkeley DB
  - Modified: keeps dir-entries in persistent memory

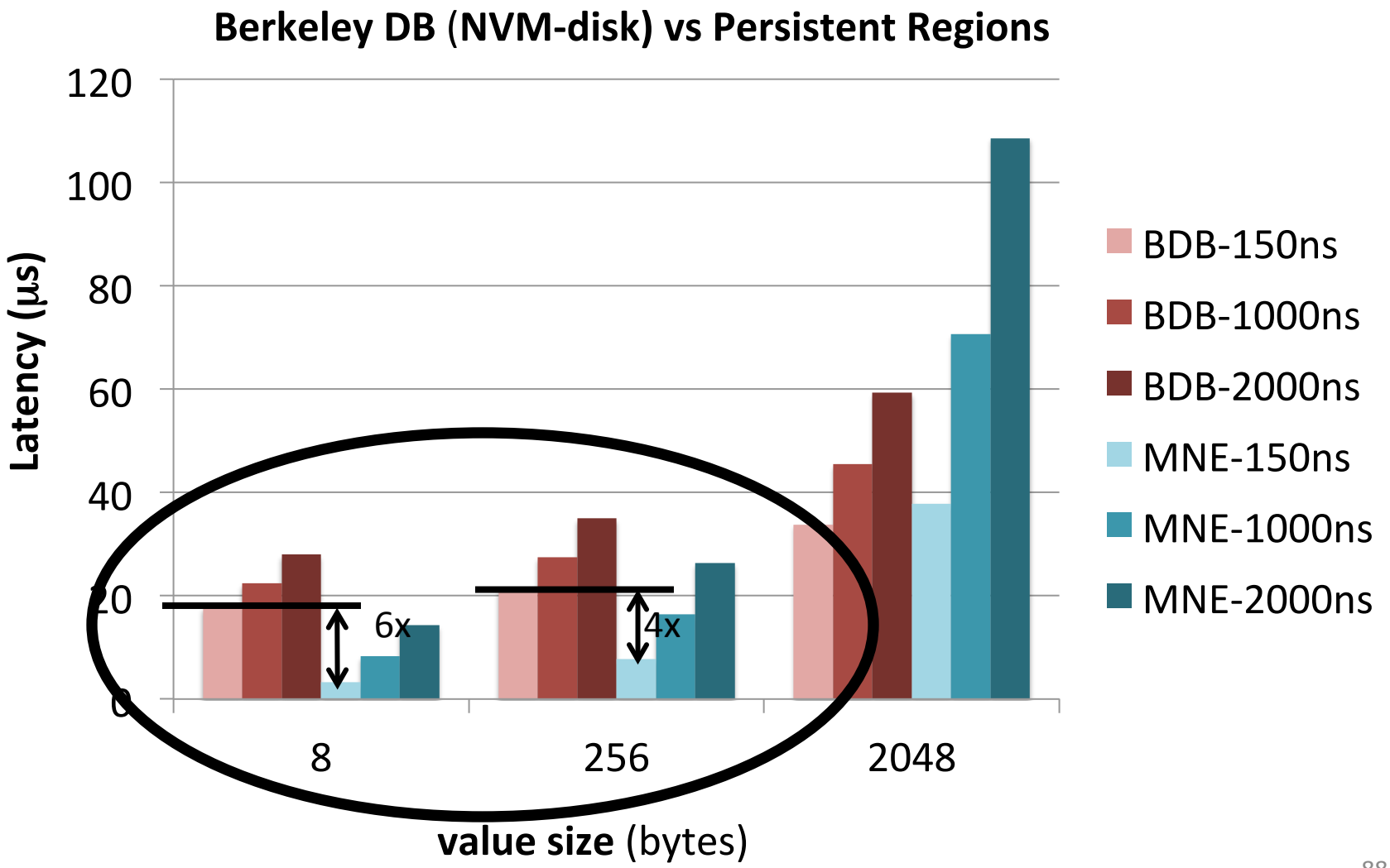
Application performance



Write-mostly workloads

- TokyoCabinet: 1024-byte ins/del queries
- OpenLDAP: template-based update queries

Hash table performance



Room for Improvement

- Software transaction overhead
- Logging doubles number of writes to NVM
- Flush/fence not yet implemented for NVM

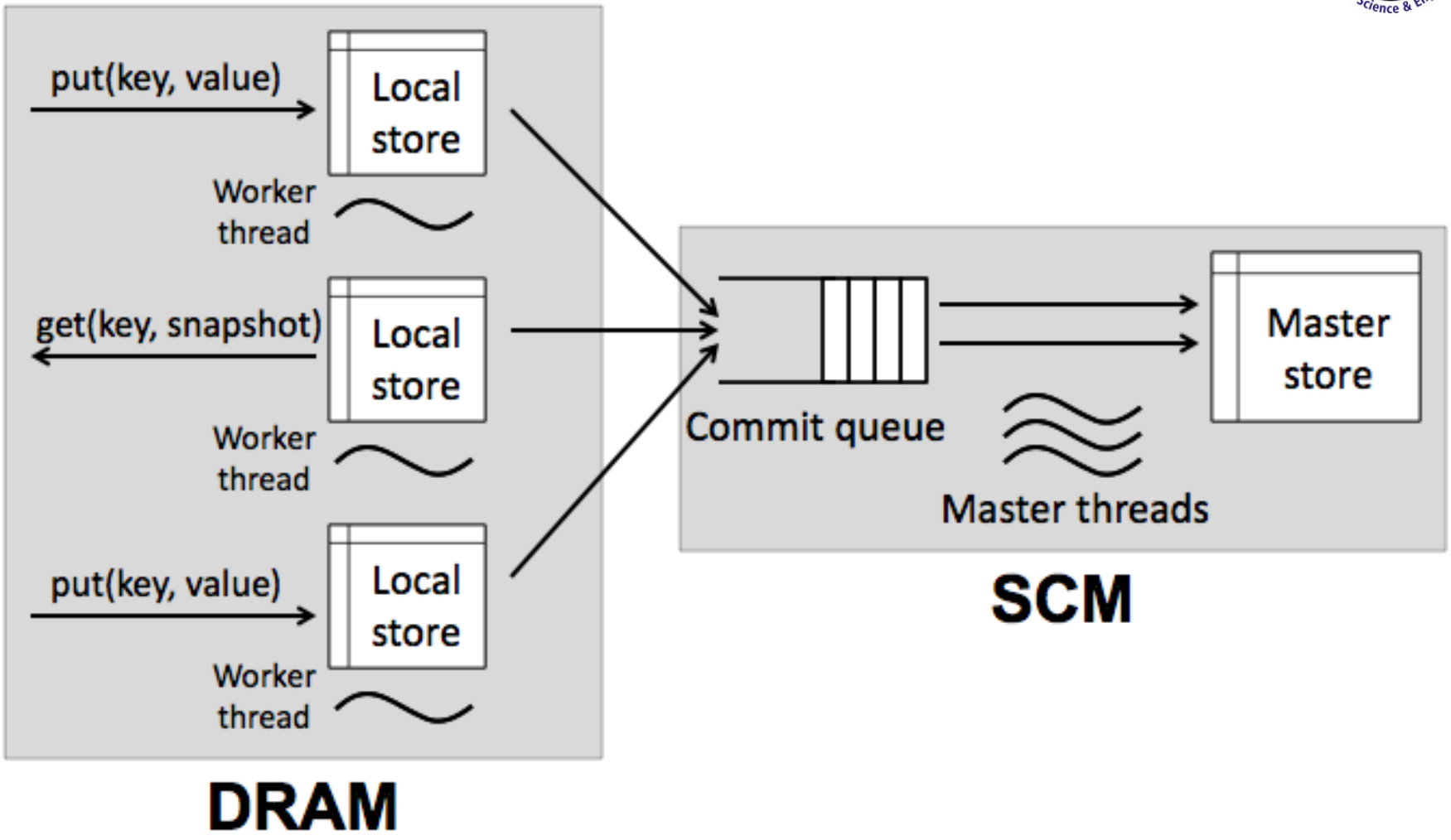
Outline

- NVM solid-state drives
- Persistent memory file systems
- Persistent Regions
- Persistent data stores

Persistent Data Stores

- Memory too low-level for programmers
- Expose key/value or object store interface

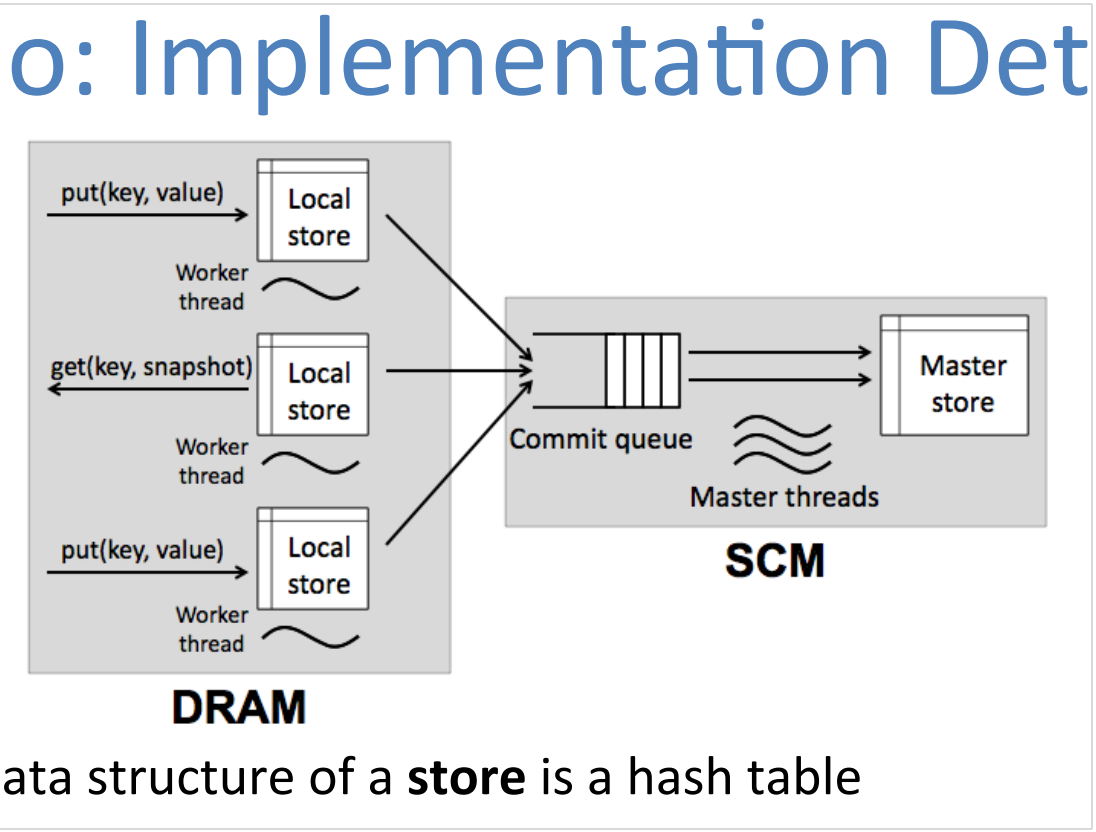
The Echo Hybrid Architecture



[Bailey, INFLOW'13]

92

Echo: Implementation Details



- The core data structure of a **store** is a hash table
- For each key, a **version table** is allocated that stores versions
- A **commit log (or queue)**, which is used to resolve conflicts and complete interrupted commits upon restart

93

Echo: Implementation Insights

- Asymmetric read/write costs
  - Writes cost more than reads and neither are as fast as DRAM

- Solution
  - Use the local stores (in DRAM) to absorb write costs until commit time

94

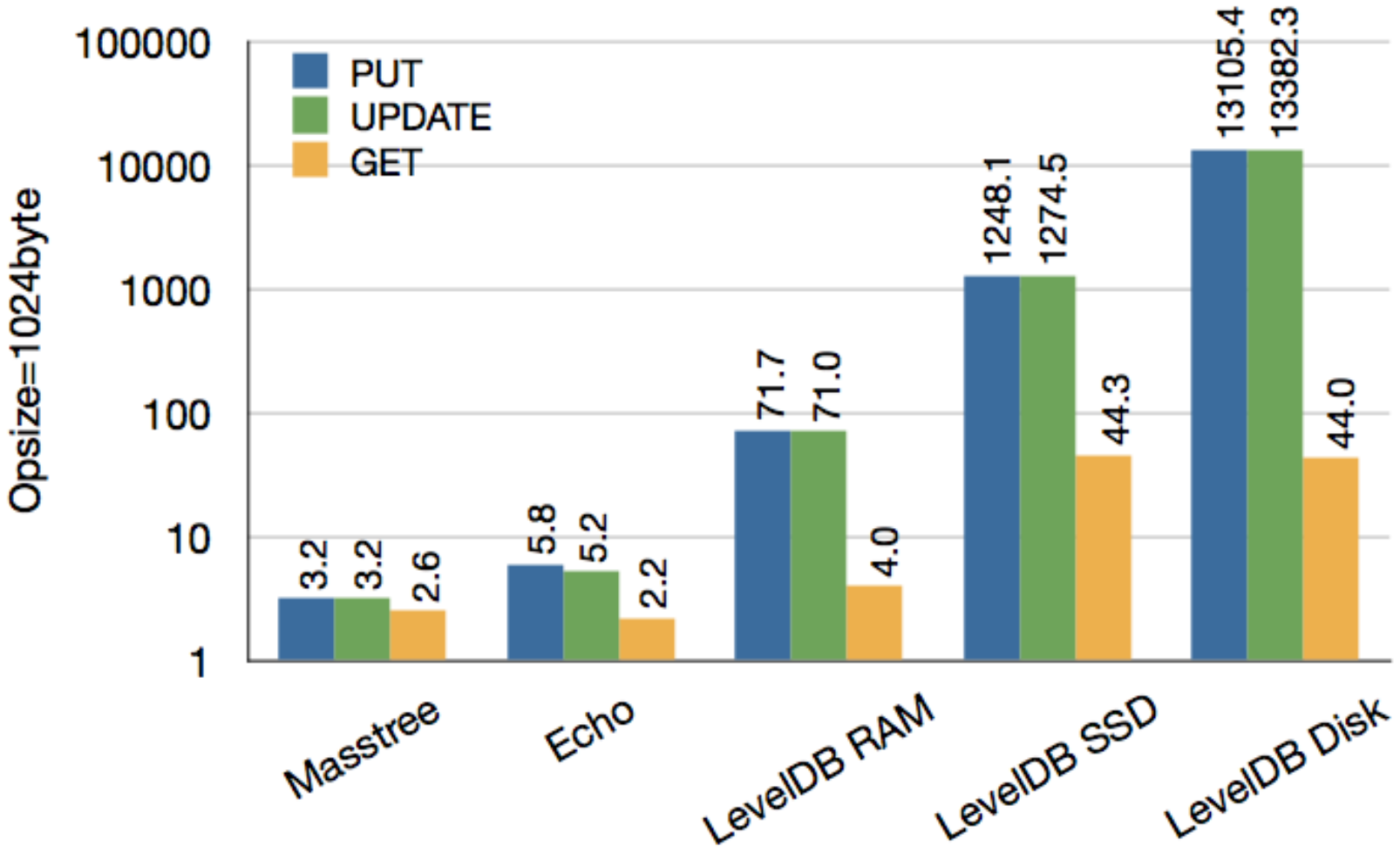
Echo: Implementation Insights

- CPU caching, as it mitigates attempts at durability
  - We lose durability for any data in the cache at power off
  - Likely creates inconsistency in the store on restart

- Solution
  - Flush important metadata and state on commit, to ensure durability
  - Flush cached SCM values on failure with a super-capacitor

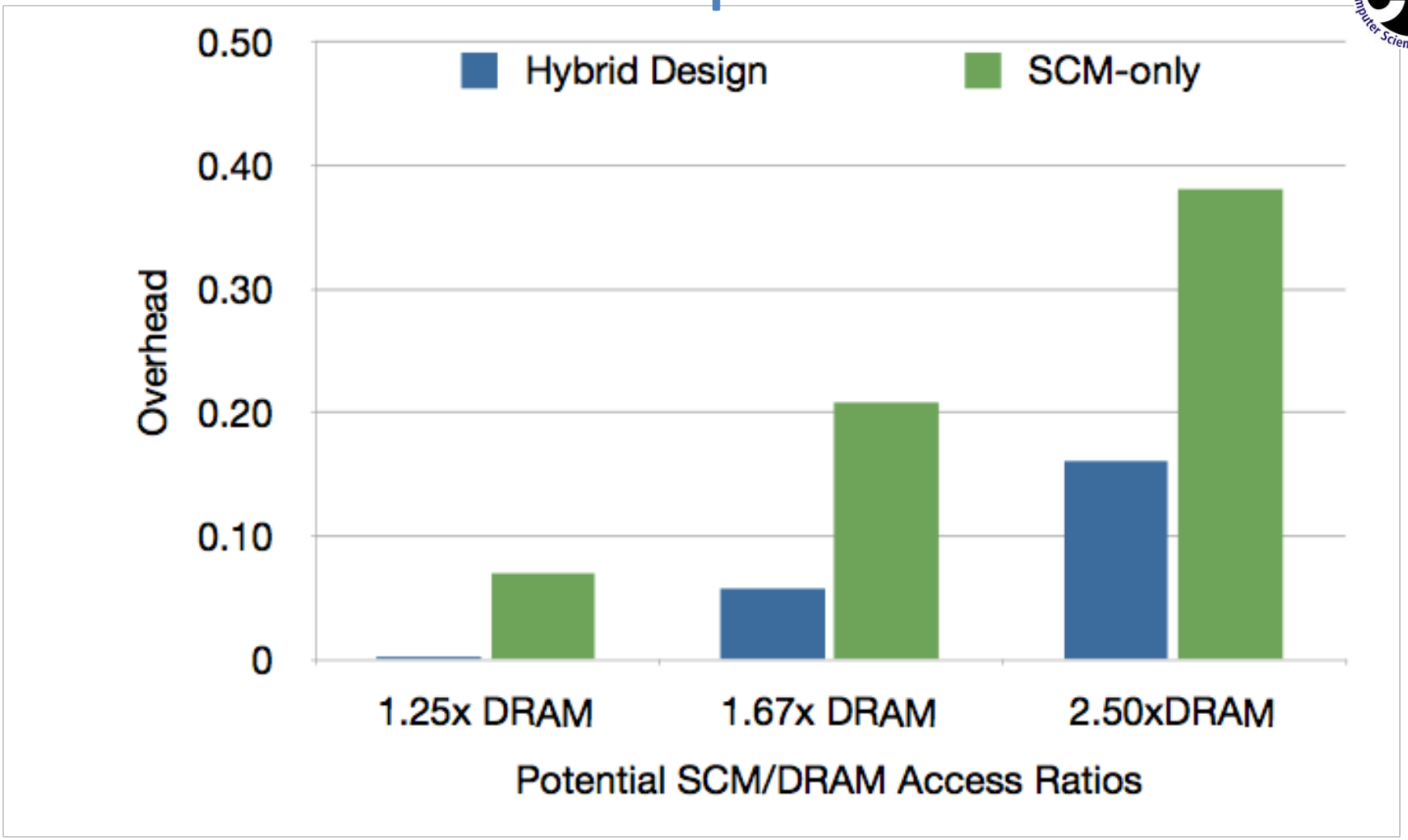
95

Echo: Base Latency



96

Echo: Impact of SCM



97

Summary

- NVM can be:
  - A fast SSD
  - Persistent memory
- Persistent memory can be:
  - A file system
  - User-mode accessible
  - Transactional
  - A key/value store

Questions?