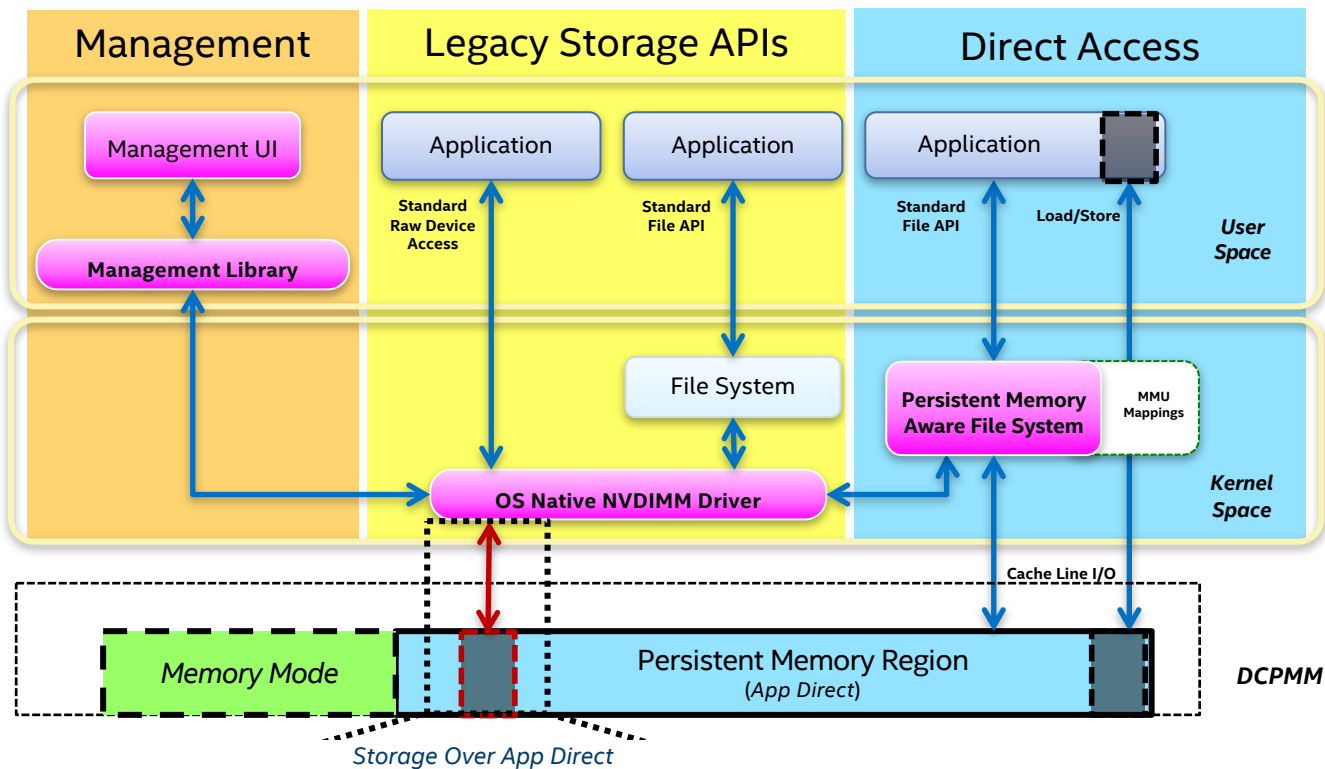




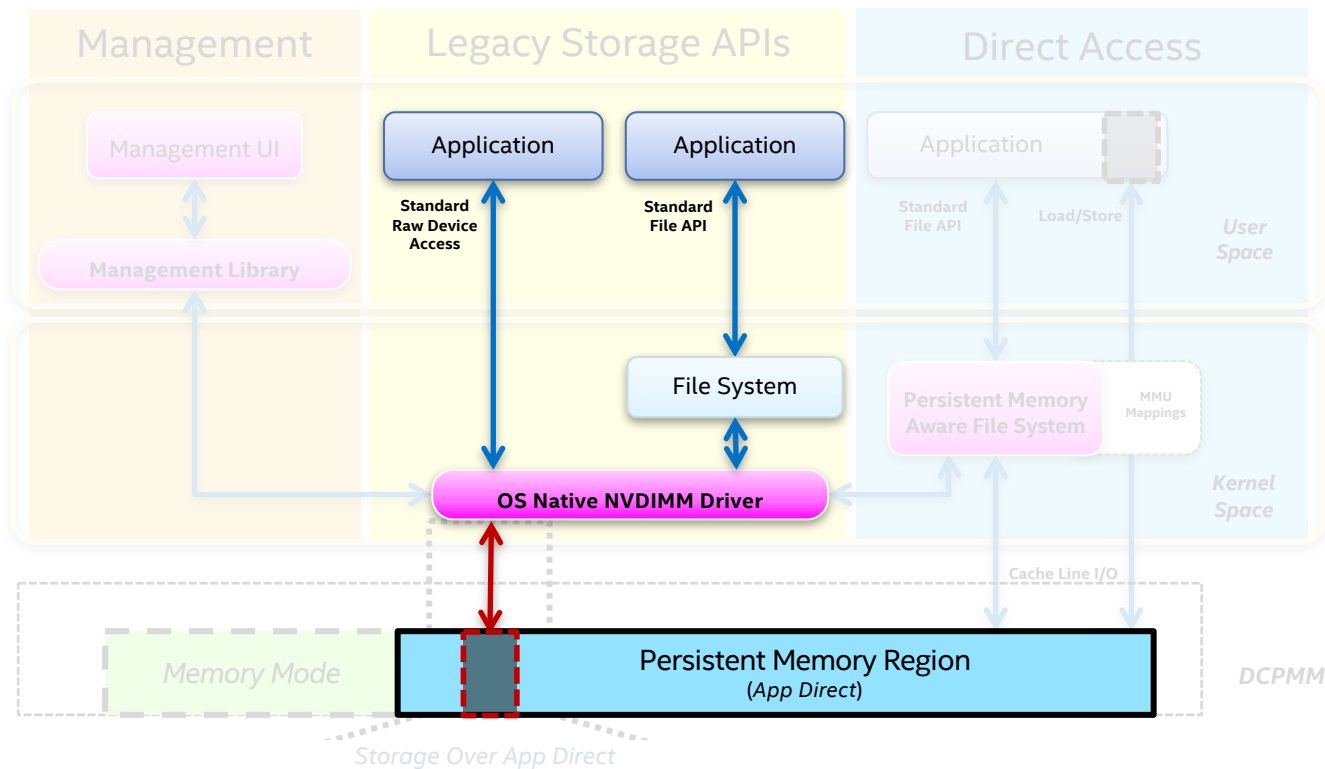
Protecting Software from Itself: Powerfail Atomicity for Block Writes

Andy Rudoff
Sr. Principal Engineer
Non-Volatile Memory Solutions
Datacenter Group
Intel Corporation

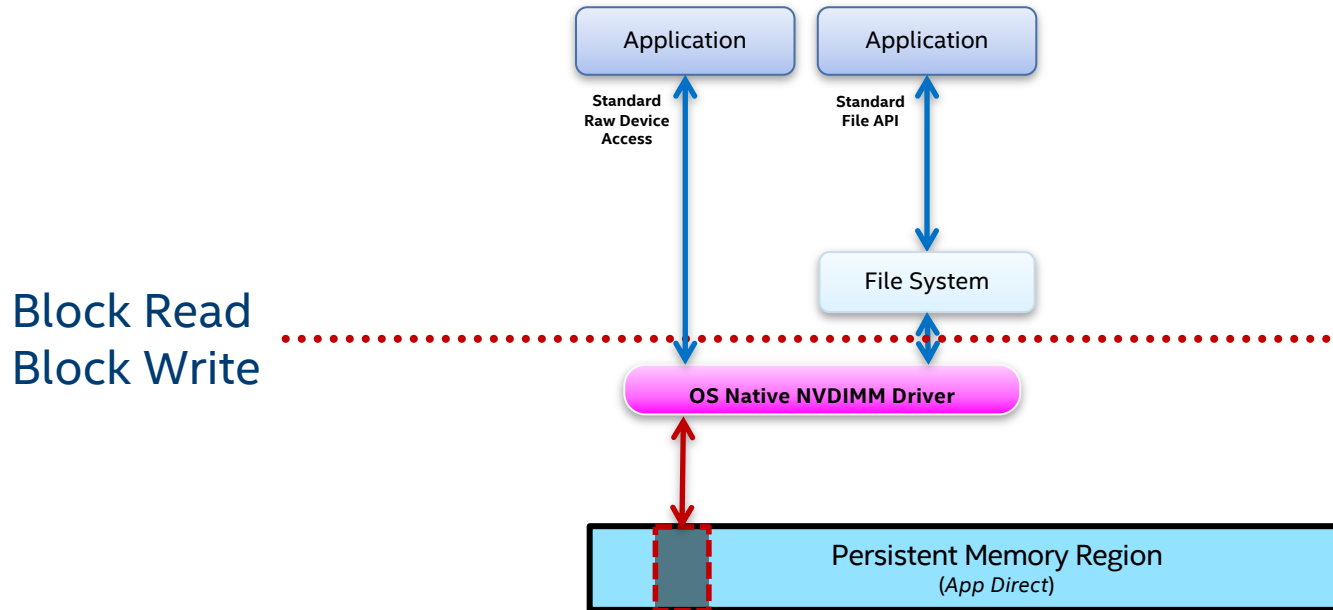
Exposing Persistent Memory to Applications



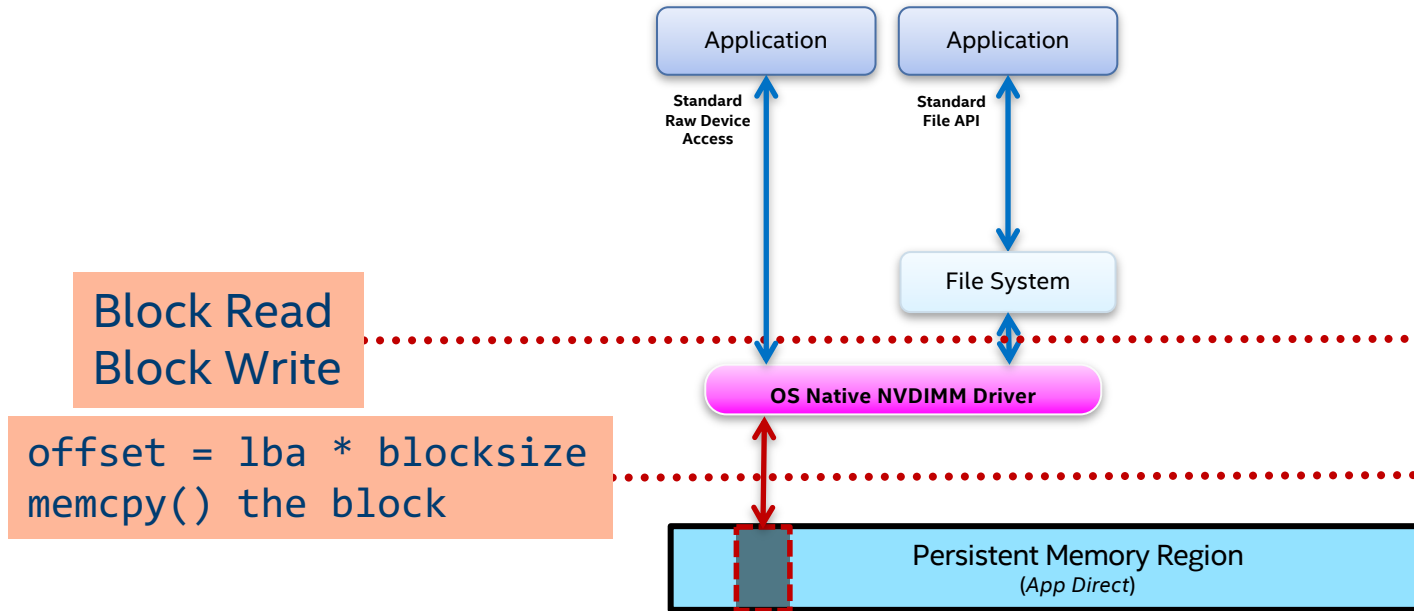
Exposing Persistent Memory to Applications



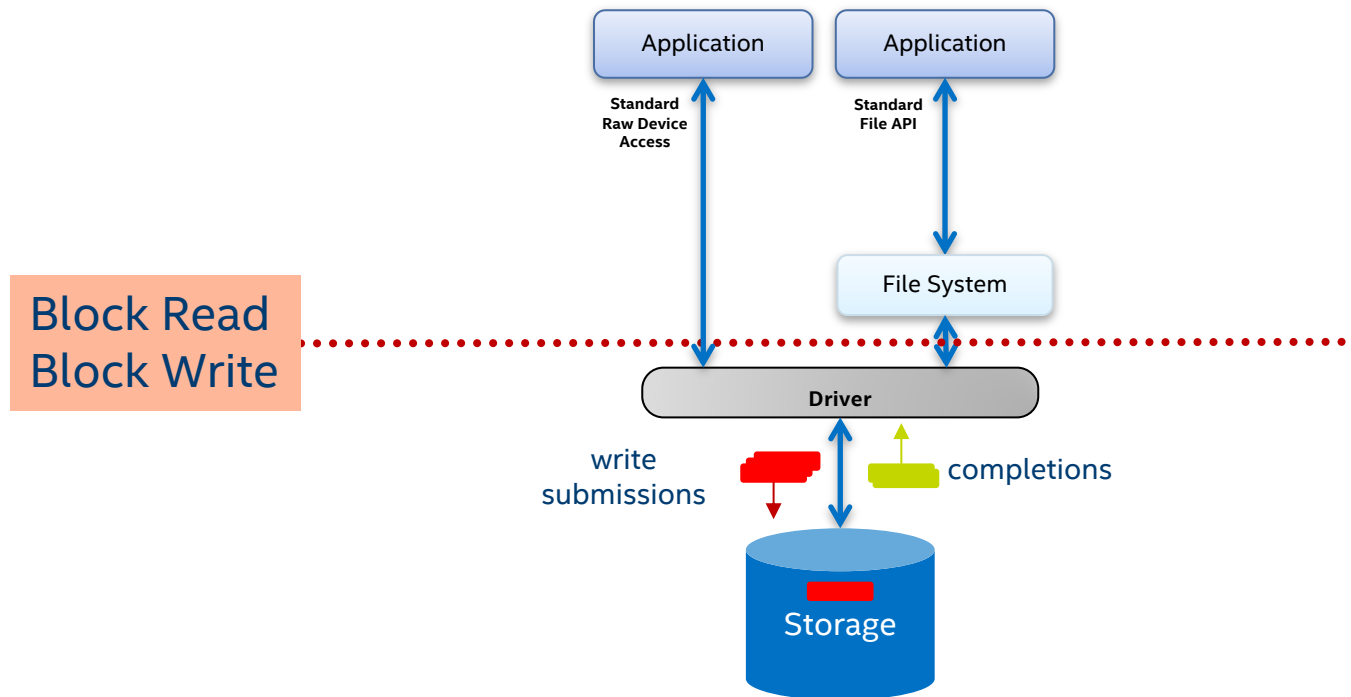
Exposing Persistent Memory to Applications



Exposing Persistent Memory to Applications



Writes In-Flight During Powerfail



After Powerfail: Hard Drives

Large write likely to be torn, but:

Individual sectors most likely to be all old data or all new data

- NOT guaranteed by spec
- Usually NOT guaranteed by manufacturer (or “best effort”)

Sector writes can be torn

- Designed to be rare
- ECC will show error, sector read returns error
- **Very rare** (bug?) part old data, part new data, NO ERROR

After Powerfail: SSDs

Large write likely to be torn, but:

Individual sectors most likely to be all old data or all new data

- NOT guaranteed by spec until NVMe, requires 1 sector not torn
- Usually NOT guaranteed by manufacturer (or “best effort”)

Sector writes can be torn

- Designed to be rare
- ECC will show error, sector read returns error
- **Very rare** (bug?) part old data, part new data, NO ERROR

After Powerfail: memcpy() to pmem

memcpy() load/store loop running on CPU interrupted

ECC is correct between any two instructions

Unlike disks, the common case is now a torn sector:

- Sector contains some old data, some new data, no read error

Rare case is non-torn sector now

Note: these torn sectors were always possible according to the specs (except for the more recent NVMe standard). Because they were rare, buggy SW has existed for decades, depending on non-torn sectors.

Fixing the Buggy SW

Several file systems depend on non-torn sectors

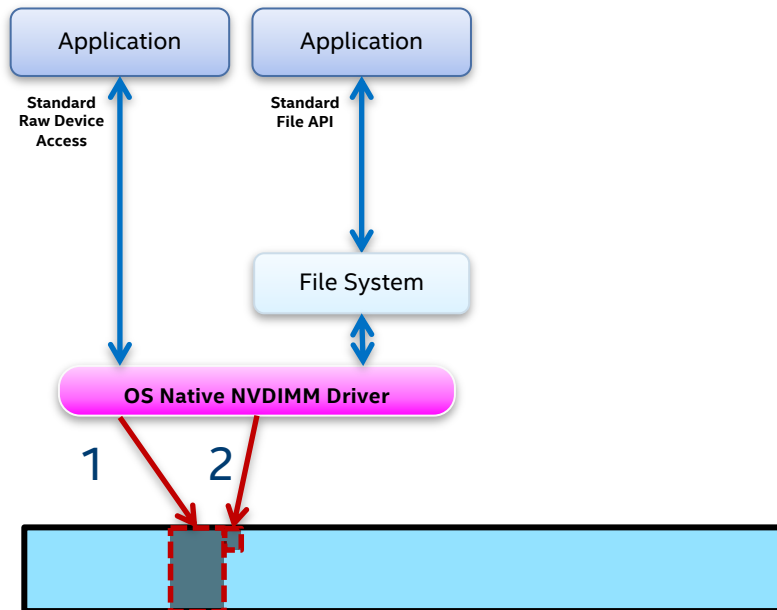
- Including NTFS

Many applications depend on it:

- Example: database writing a sector to a log file, unprotected by checksum

Result: implement powerfail atomic sectors on pmem

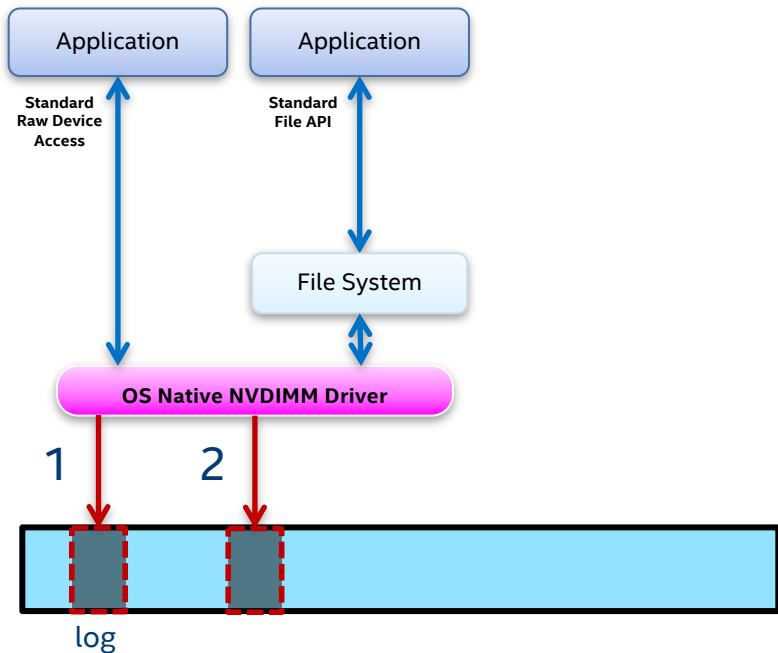
First Approach: Detect Torn Write



1. Write block
2. Write checksum

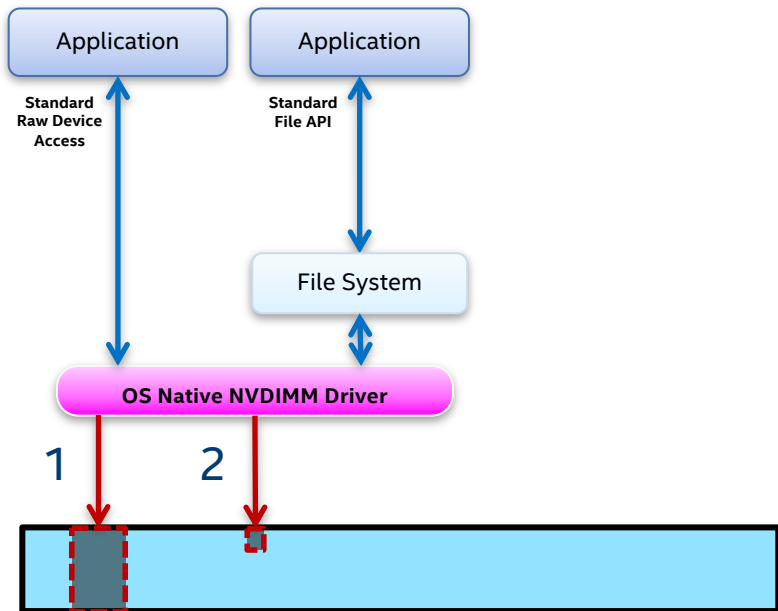
- Virtually all in-flight writes return error

Better Approach: Write-Ahead Logging



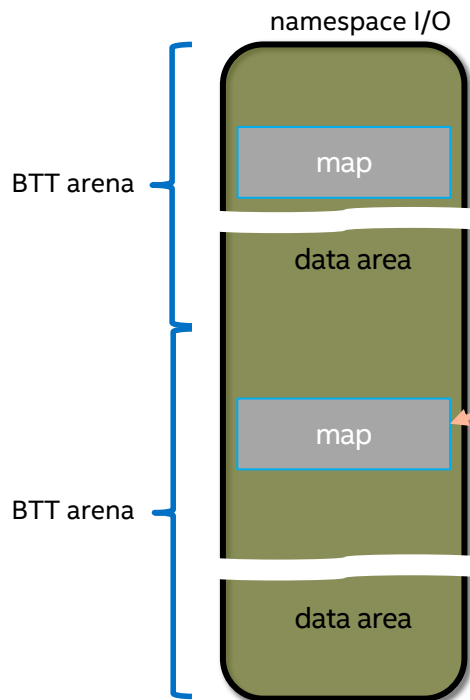
1. Write block to log
 2. Update actual LBA in-place
 3. On Recovery:
 - Remove incomplete log entries
 - Finish any partial LBA updates
- Write amp of 2
 - A little more due to metadata

Better Approach: All Writes are Allocating Writes



1. Write data to free block
 2. Update map/free list
 3. On Recovery:
 - Finish metadata updates
- Reads require consulting map
 - One 32-bit load added per read
 - Write require metadata update
 - One load, two stores,
 - Plus cache flushes
 - Larger blocks amortize overhead better

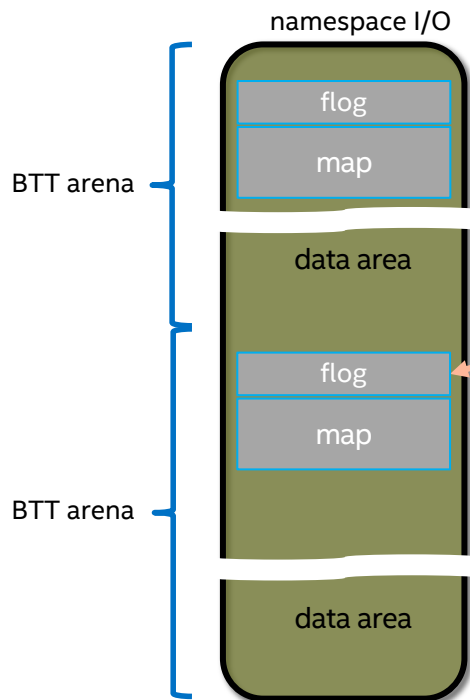
Adding a Level of Indirection: The Map



Map shows current location of each LBA

- Index into map by LBA number
- Get “internal LBA” back
- Metadata opportunity:
 - Zero bit
 - Error bit
- Updates **atomic**
- 4-byte entries
 - “arenas” make this possible

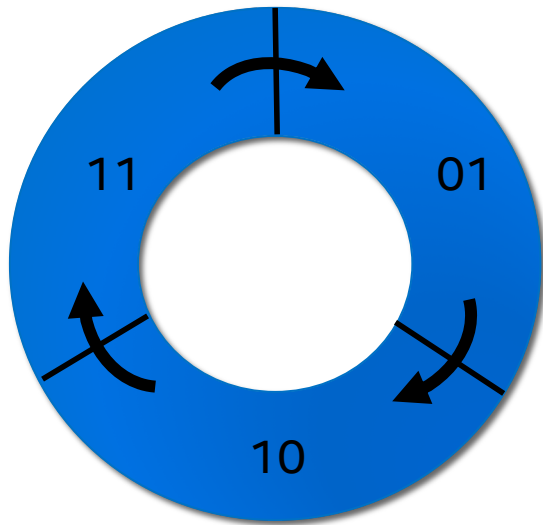
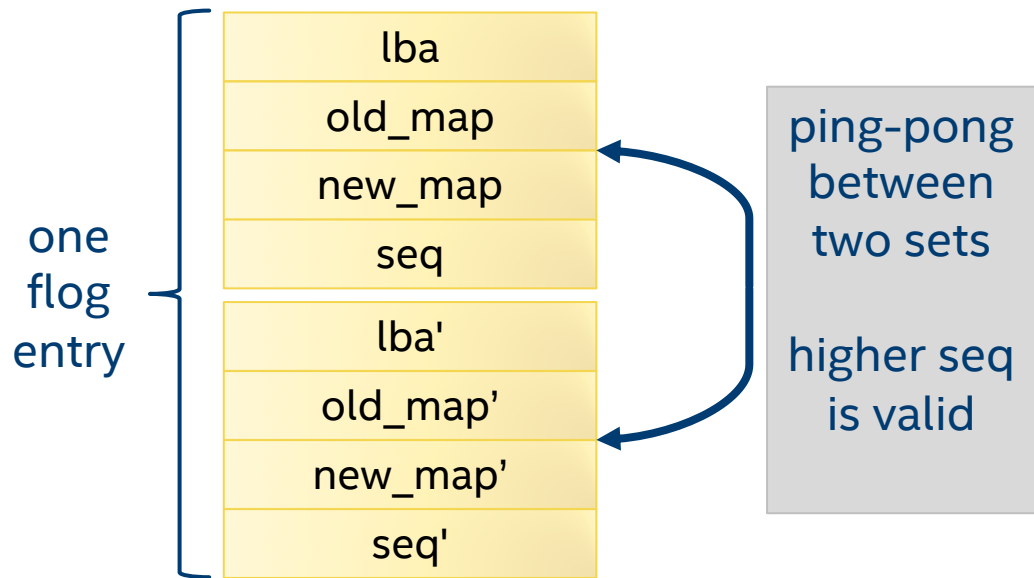
Managing the “free list” – log style



Flog tracks free blocks

- 1 block per possible in-flight write (NCPU)
- A write goes to a free block, then
- Update flog entry (**atomic**)
 - Updates which block is free
 - Contains info to update map
- Update map
 - Finished on recovery is needed

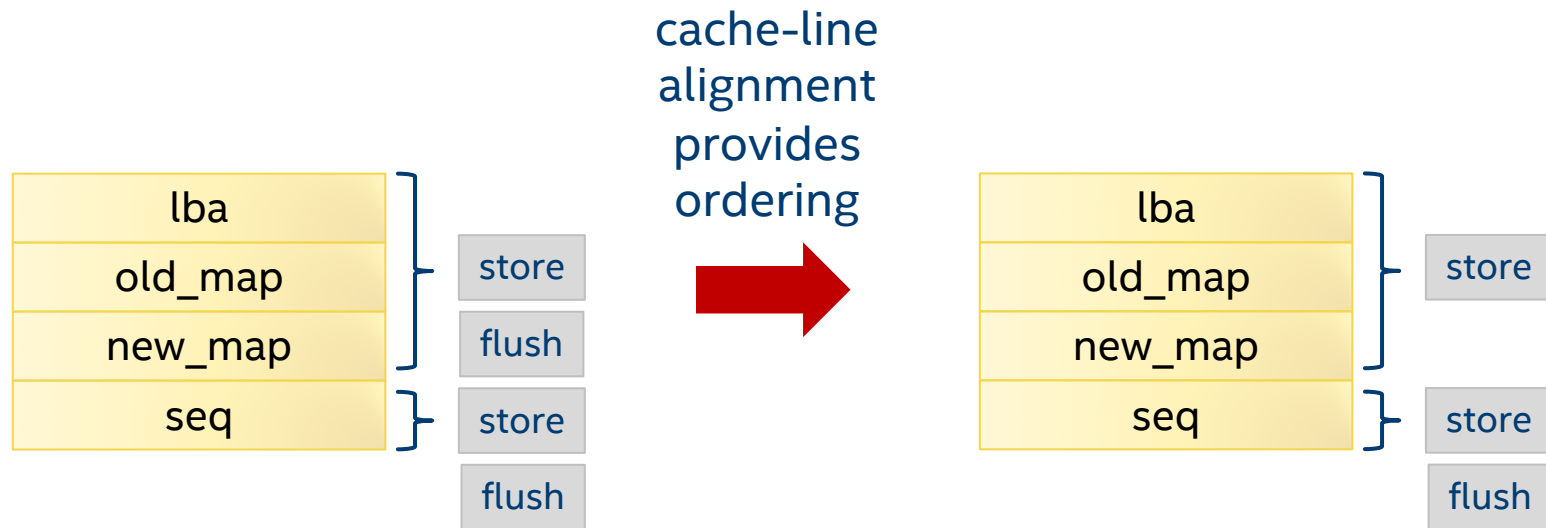
Atomic, Efficient Flog Updates



Making Sure seq Field is Persistent Last

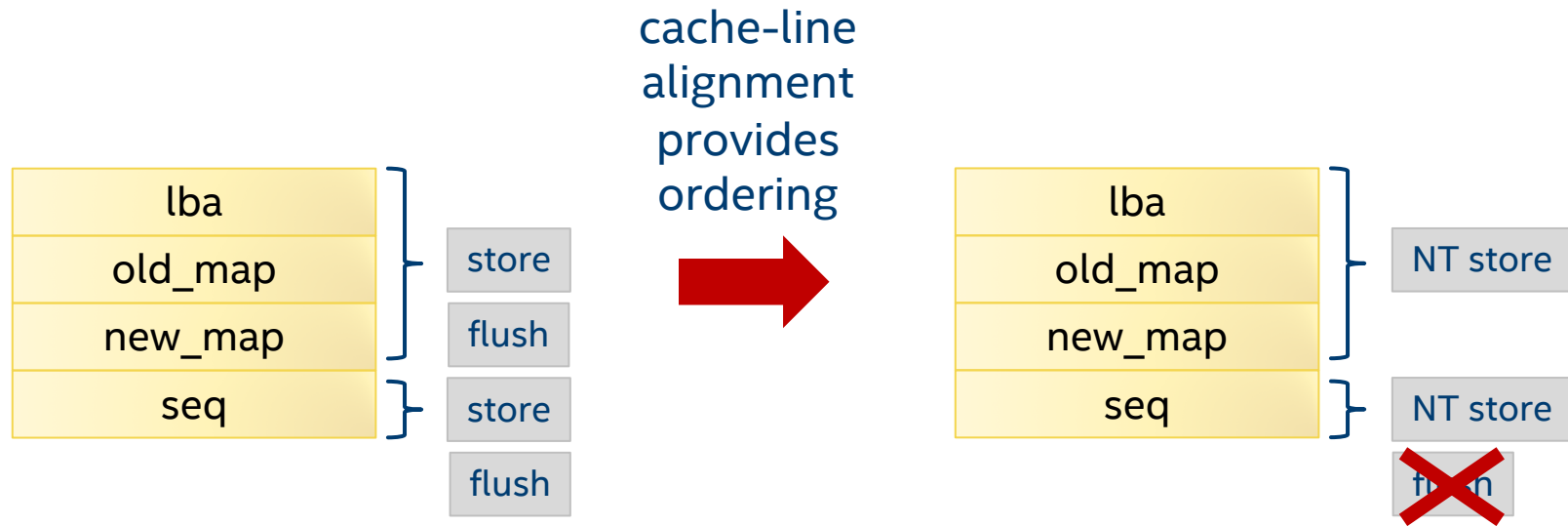
```
update_flog(...)  
{  
    ...  
    flogp->lba = lba;  
    flogp->lba = old_map;  
    flogp->lba = new_map;  
    pmem_persist(flogp, sizeof(uint32_t) * 3);  
    flogp->seq = seq;  
    pmem_persist(&flogp->seq, sizeof(uint32_t));  
}
```

Atomic, Efficient Flog Updates on x86



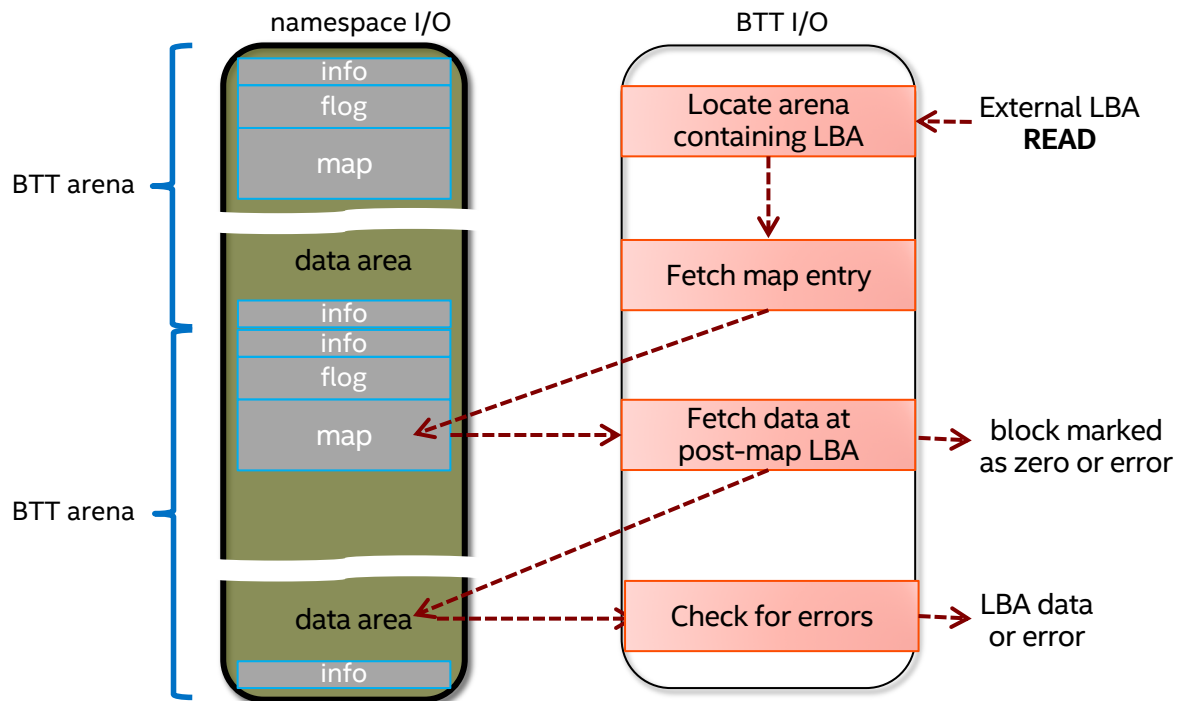
seq field becomes persistent last, atomically making all fields valid

Atomic, Efficient Flog Updates on x86

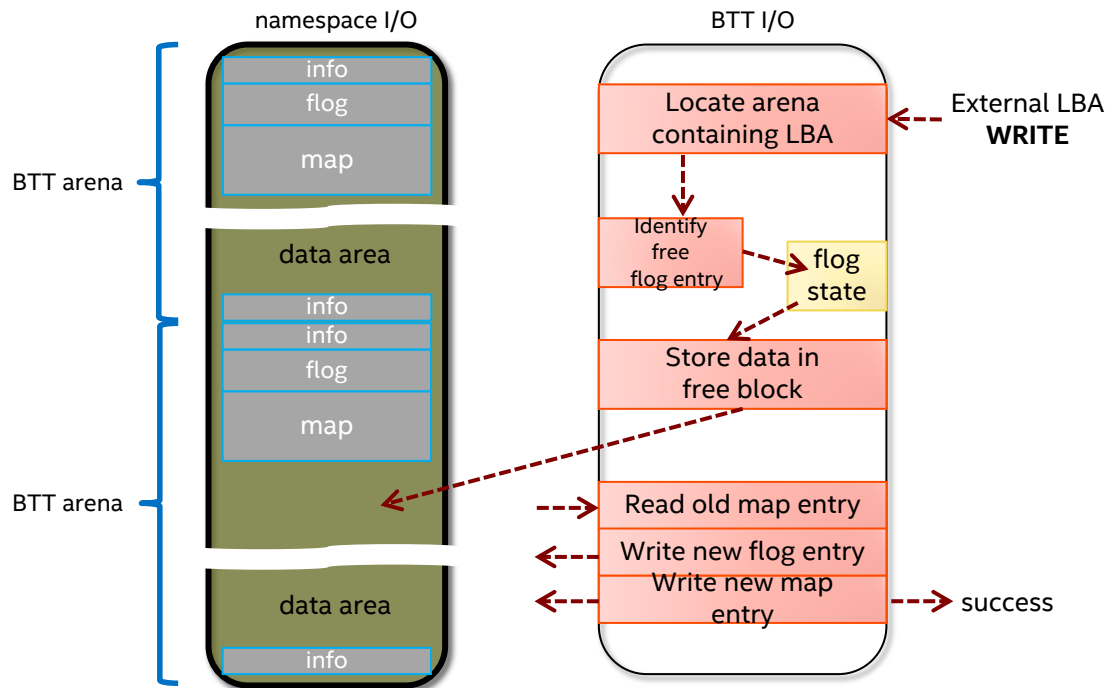


seq field becomes persistent last, atomically making all fields valid

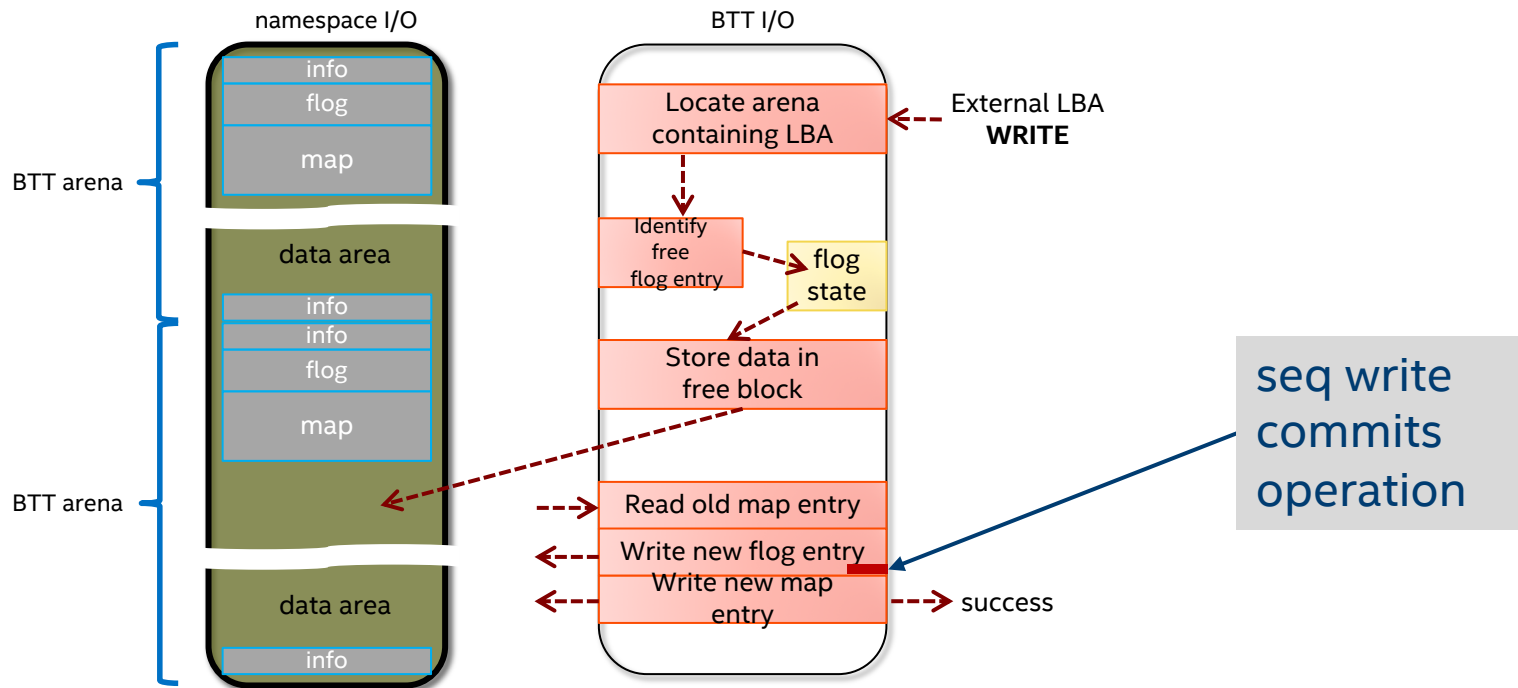
Read Path



Write Path



Write Path



Adding BTT to the Ecosystem

Reference implementation: libpmemblk

- Code used in multiple OS implementations

Standardized

- uefi.org (UEFI chosen since BTT used during boot)

Performance

- Not terrible, but not trivial (15-20% write overhead)
- Raised awareness of write atomicity issue
- Some file systems fixed/enhanced so BTT isn't needed for that FS