Improving I/O on fast, byte-addressable NVM

Haosen Wen

Oct. 15, 2021

Goal

- Low-latency, high-bandwidth I/O
- Necessary protection against bugs and failures
- Easy to use

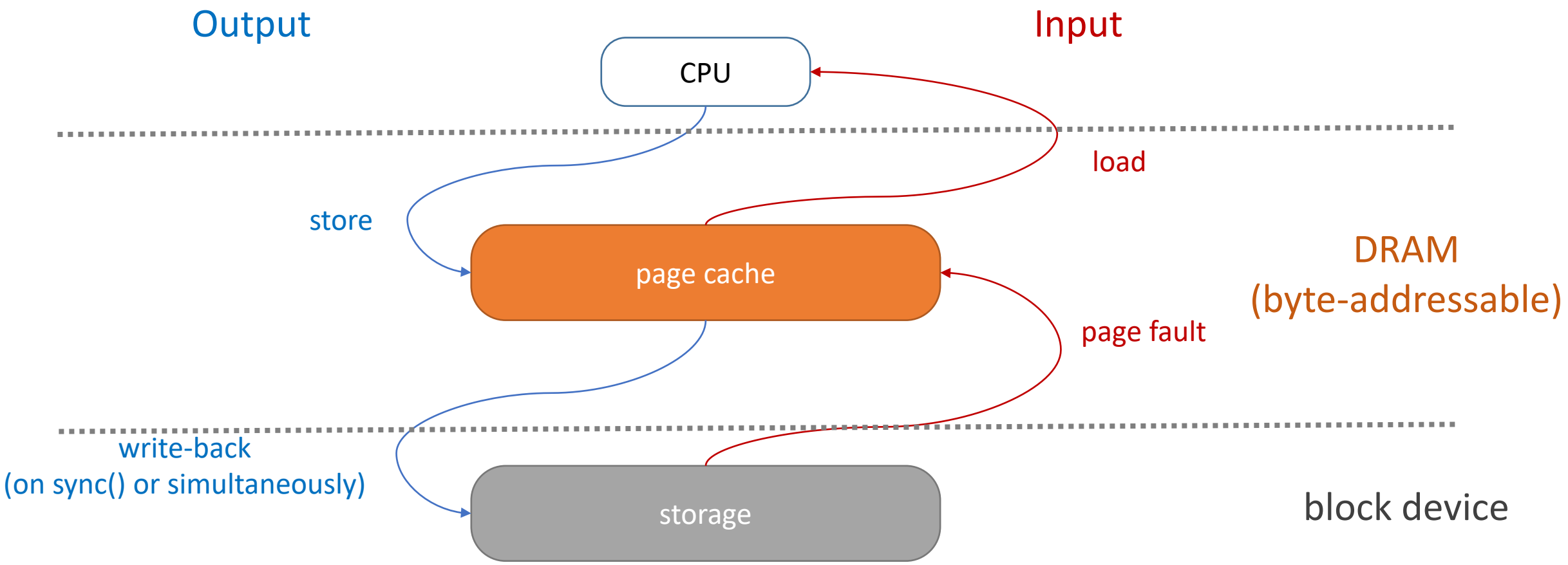
Outline

- NVM in local synchronous I/O
  - Memory-mapped I/O
  - POSIX-style file operations
- Distributed storage on NVM
  - Asynchronous I/O?
- Reduce kernel intervention I/O on NVM
- Some mixture of all

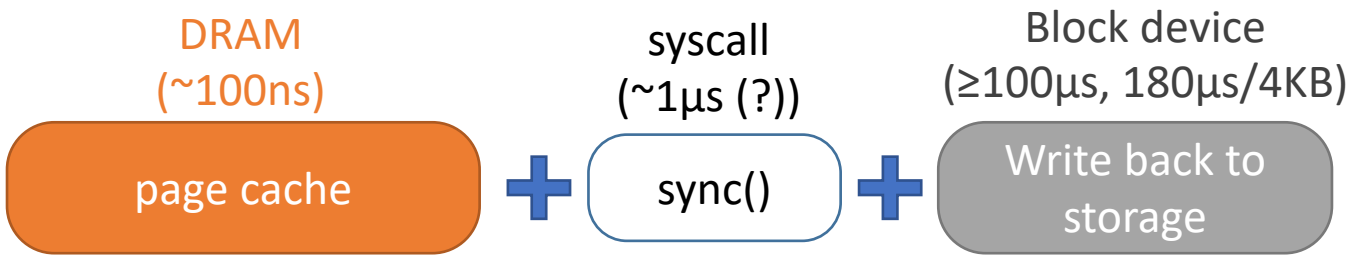
I/O on block devices

- POSIX-style file API
  - open(), seek(), read() and write() sys calls to access files
  - Interfaces like AIO for asynchronous accesses
  - File system checks access privilege of application(s) for protection
  - Easier to use: failure atomicity provided by FS; serialization of write()'s
- mmap-ed I/O
  - mmap() maps a (chunk of a) file to the virtual address space of an application
    - Virtual address space is private to each process
  - Access with user-level store and load instructions, (almost) always synchronous

Memory-mapped I/O (Linux)



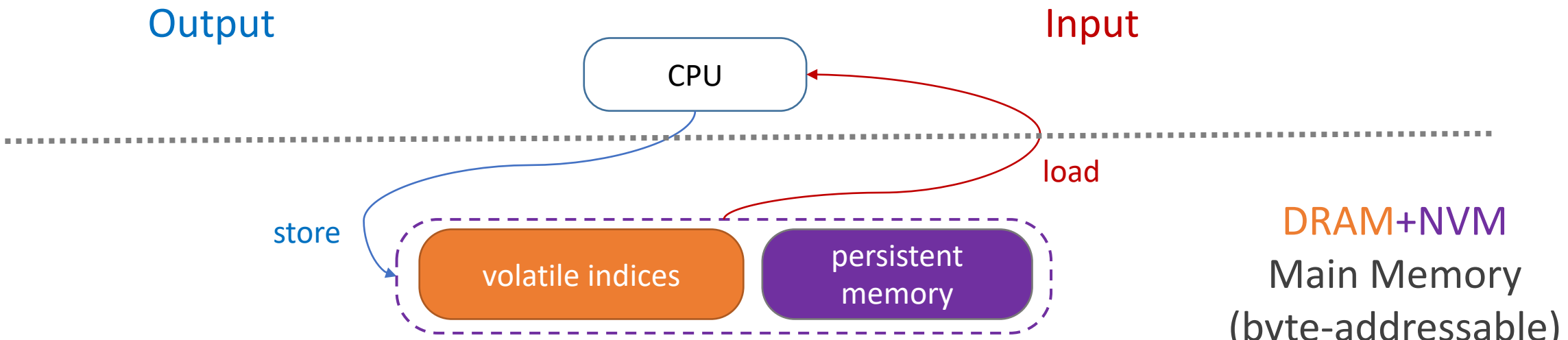
Latency of persistence



- Note: page cache may get written back at the convenience of kernel/FS

Reference: https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_2104.pdf

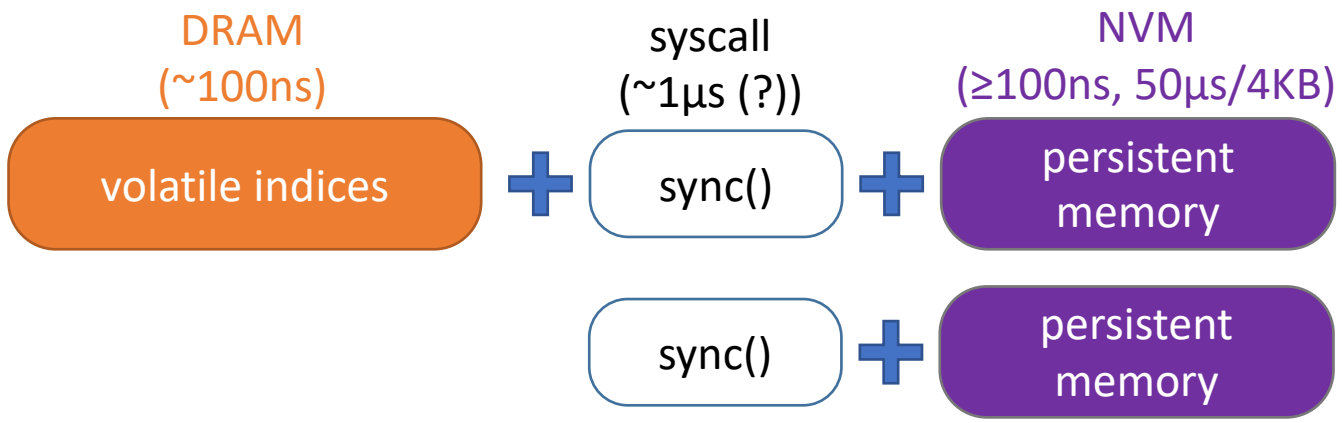
Memory-mapped I/O (Linux)



- With file systems supporting Direct Access (DAX-)mmap (e.g. Ext4-DAX, XFS), CPU can access NVM with store and load instructions.
- *Failure atomicity is non-trivial*:
  - Failure-atomic msync[Park et al., EuroSys'13], atomic mmap[Xu and Swanson, FAST'16]
  - General purpose systems: PMDK, persistent STMs, iDo, Montage, etc.
  - Dozens of ad-hoc persistent data structures

less general

Latency of persistence



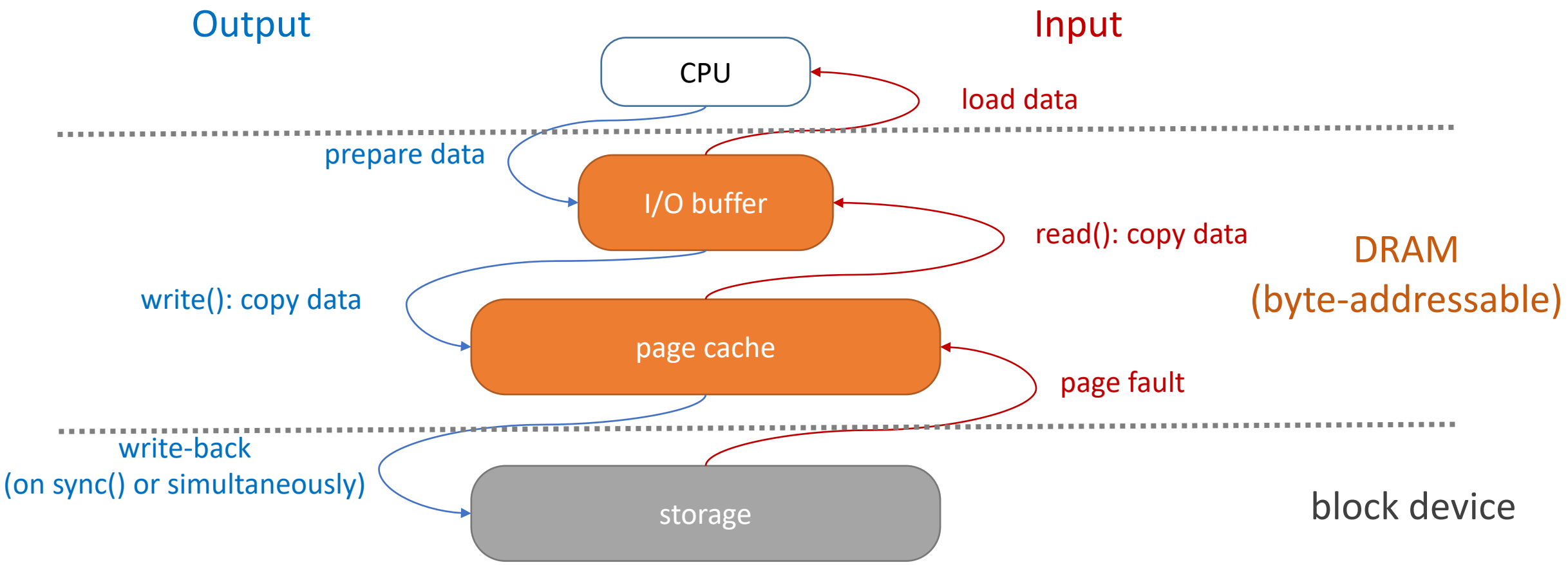
- Probably the best we could hope for
- Only fits applications implementing memory-mapped I/O
  - Without sophisticated infrastructures for NVM, sync() and non-temporal store/load are usually the only tool to guarantee persistence

Data Reference: Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, Izraelevitz et al., ArXiV, Aug 2019

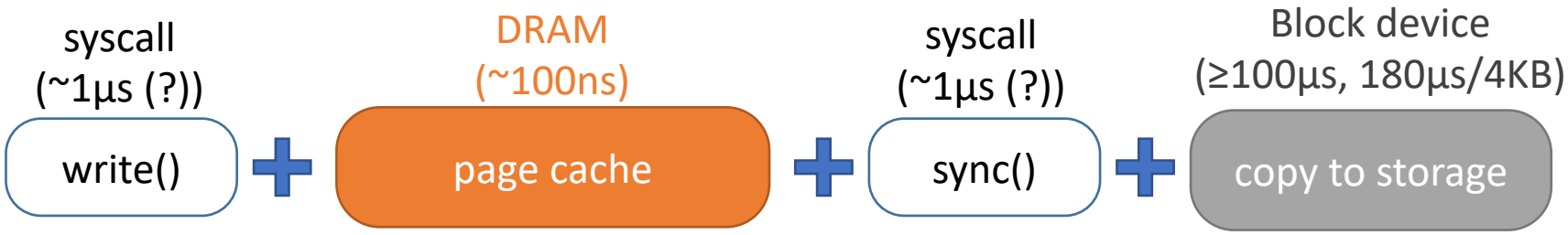
Memory-mapped I/O on NVM

- Nova[Xu and Swanson, FAST'16] allocates copy-on-write replica pages for atomicity
  - It copies replica data back to the original on msync()
  - Replica pages are always on NVM
- Are these really the best choices?
  - Remap pages to replace original with replica? (nvramdisk[Jung et al., ToC'16])
  - Put (some) replica pages in DRAM?

POSIX file operations (Linux)



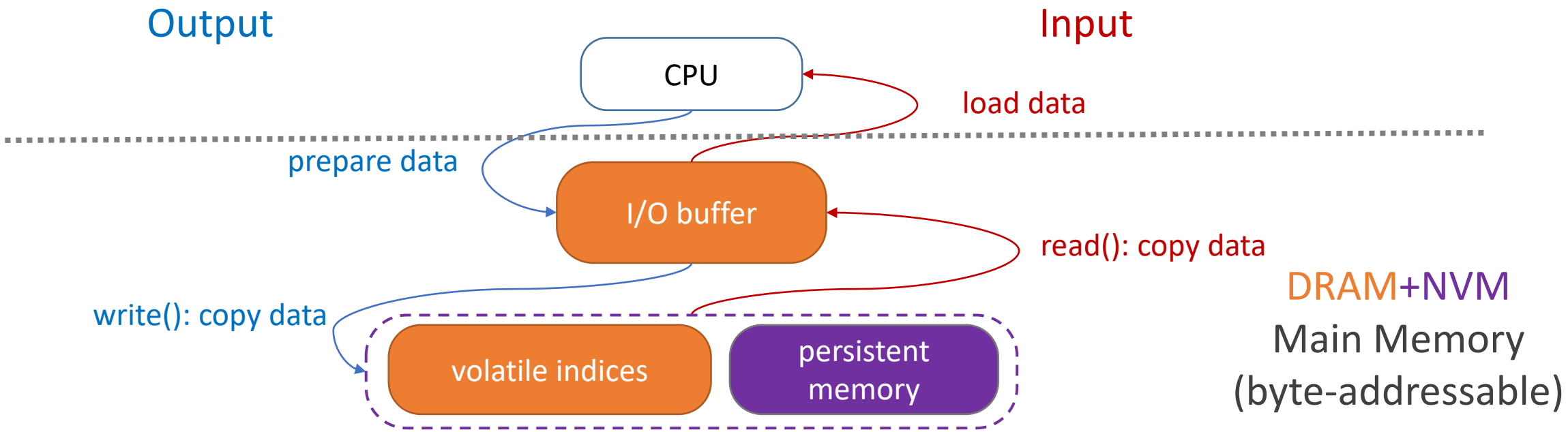
Latency of persistence



- Note: page cache may get written back at the convenience of kernel/FS

Reference: https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_2104.pdf

POSIX file operations (Linux)



- This model is picked up by most existing NVM file systems when implementing POSIX-compatible interfaces
- Why is I/O buffer still there?

POSIX read()/write() examples in C

```c
/* write() */
char buf[128];
fd1 = open("foo", O_WRONLY |
        O_CREAT);
scanf("%127s", buf);
write(fd1, buf, strlen(buf));
close(fd1);
```

```c
/* read() */
char buf[128];
fd1 = open("foo", O_RDONLY |
        O_CREAT);
read(fd1, buf, 128);
close(fd1);
```

POSIX read()/write() interface *implies* copy-based data transfer [Kim et al., APSys'20]

Reference: https://en.wikibooks.org/wiki/C_Programming/POSIX_Reference/unistd.h/write

peek() and patch() [Kim et al., APSys'20]

/ write() equivalent */*

```
void* buf = alloc_pmem(…);
fd1 = open("foo", O_WRONLY |
        O_CREAT);
scanf("%127s", *buf);
patch(fd1, buf, strlen(buf));
free_pmem(buf);
```

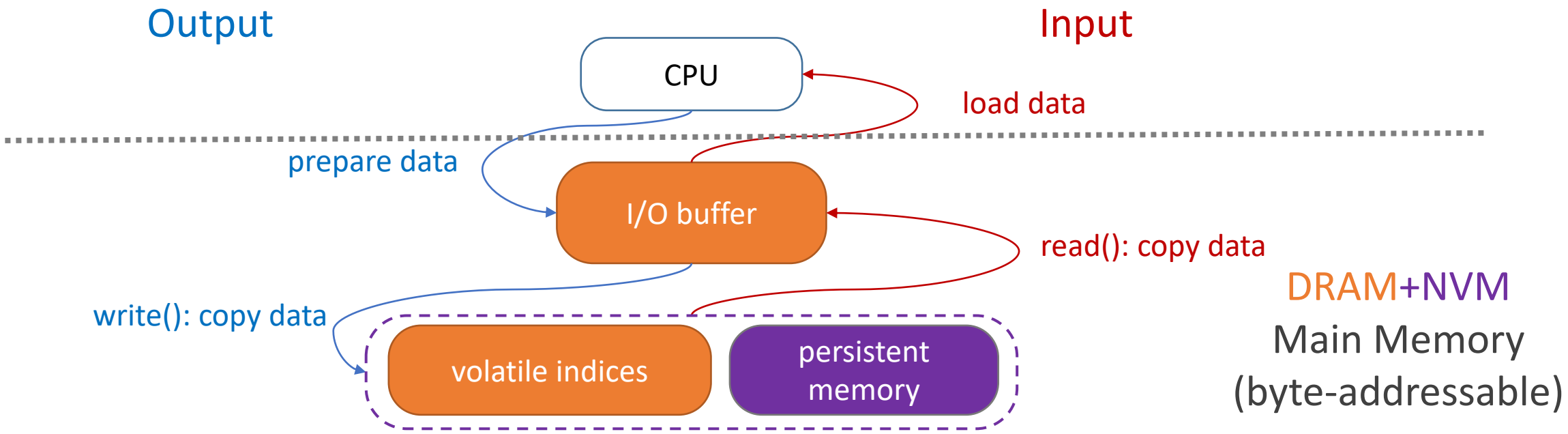
No copy; page table update only

/ read() equivalent */*

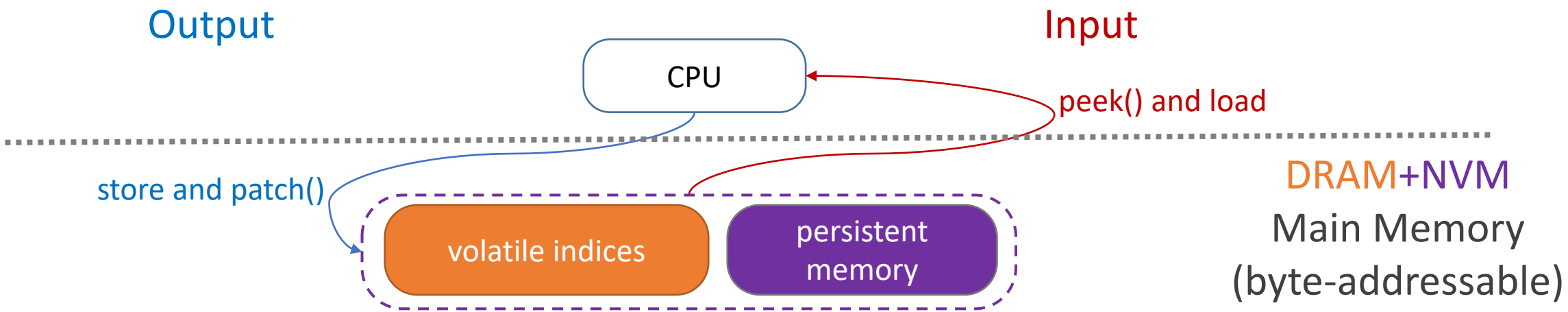
```
fd1 = open("foo", O_RDONLY |
        O_CREAT);
char* buf = peek(fd1, 0, in_len);
…
unpeek(buf);
```

No copy; returns reference only

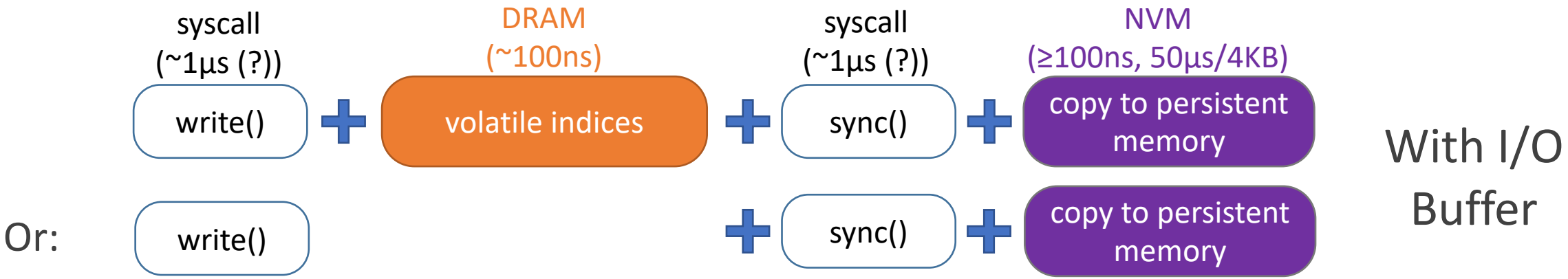
POSIX file operations (Linux)



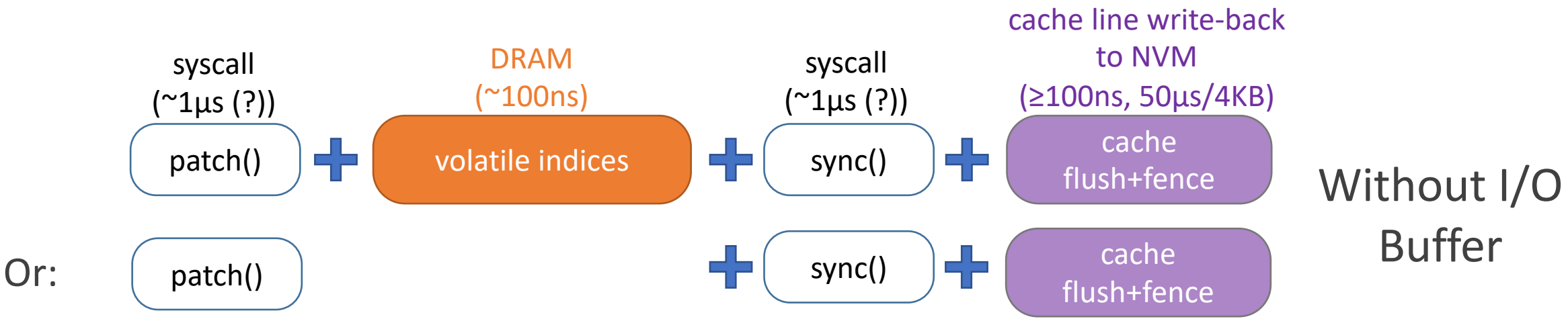
POSIX file operations (Linux)



Latency of persistence

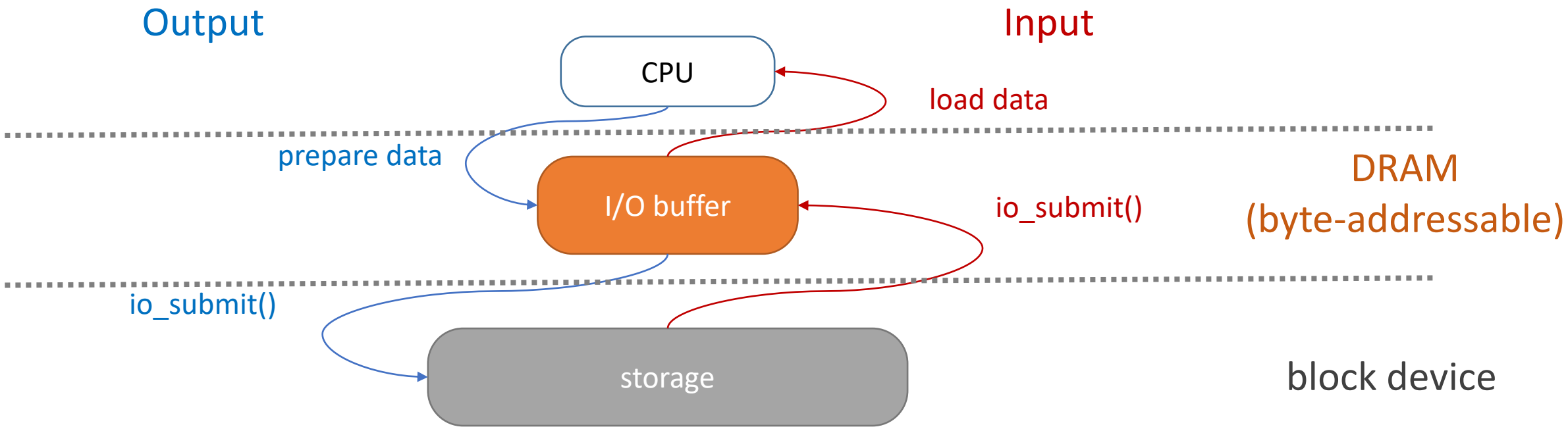


Latency of persistence



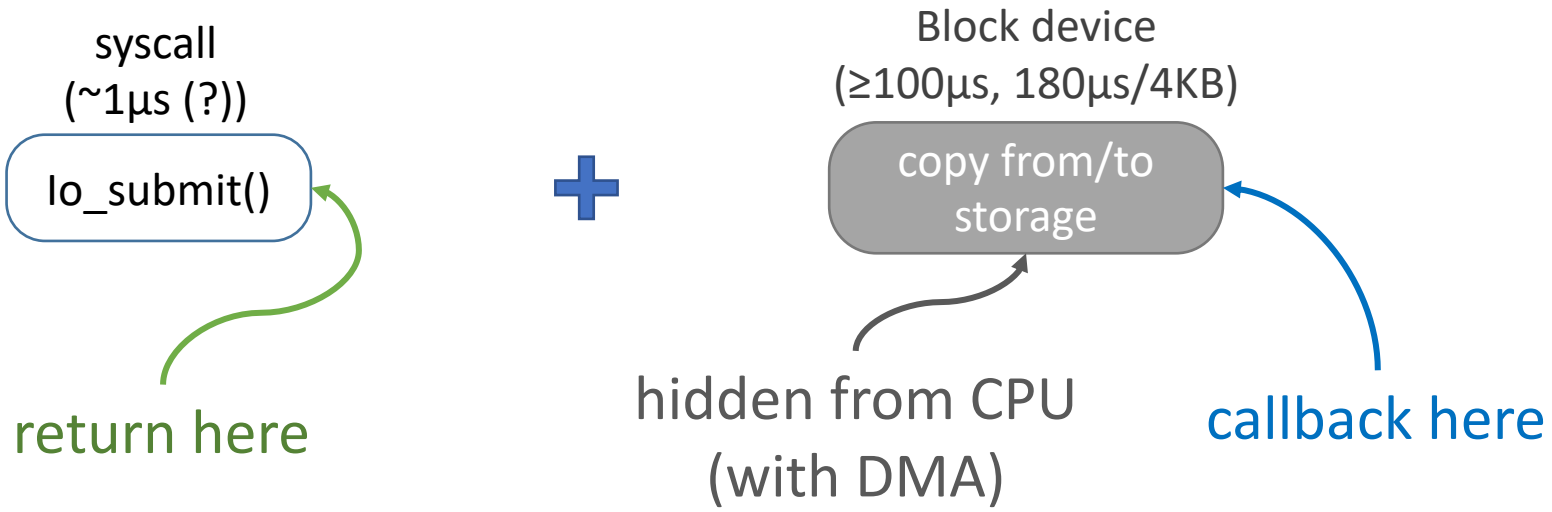
- This may be “cheating”: we are not counting in the additional latency when preparing the patch directly on NVM rather than DRAM
- By far the simplest mechanism of NVM-based file interface

Asynchronous I/O with DMA



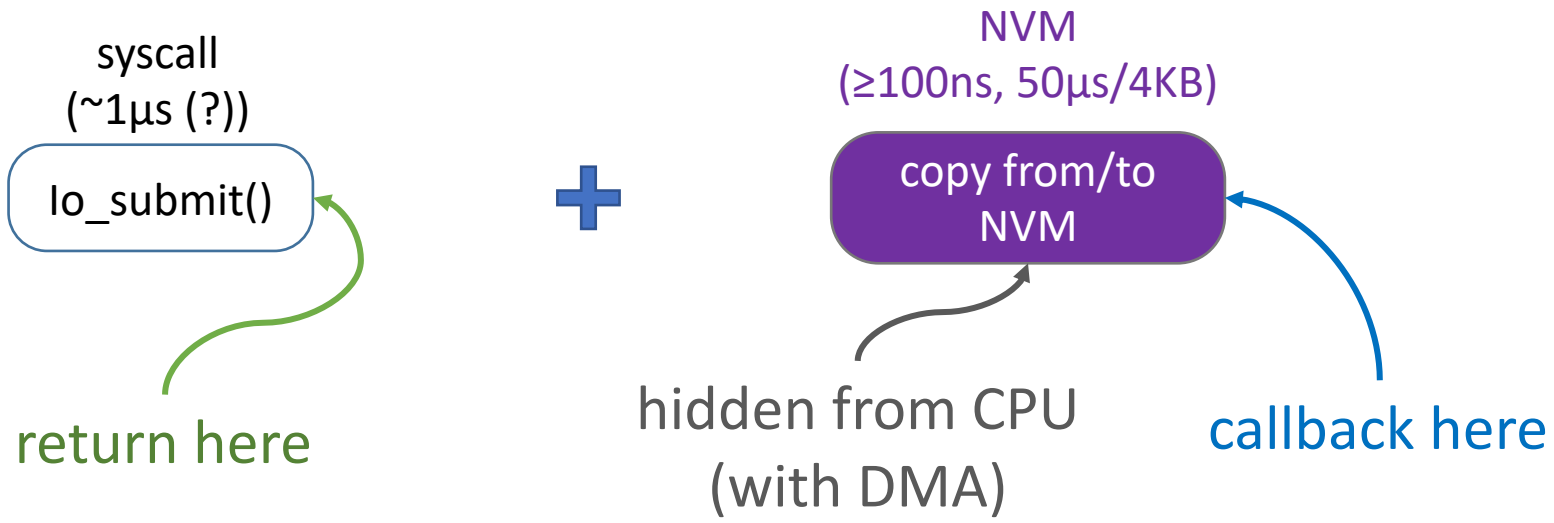
- io_submit() returns when the I/O request is *initiated*. When I/O finishes, a callback function is called.

Latency of persistence



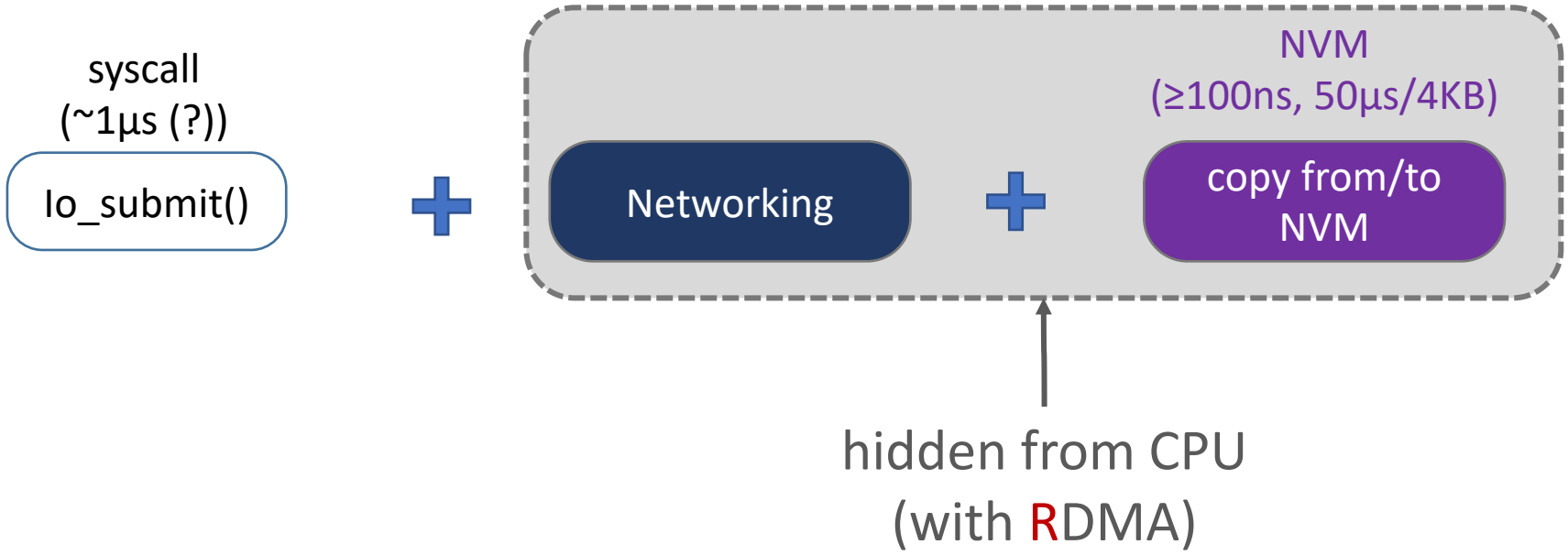
- Very useful when storage latency is high and/or bandwidth is low

Latency of persistence



- Very useful when storage latency is high and/or bandwidth is low
- With NVM, nothing much to hide ┓(̄Д ̄)┏

Latency of persistence



- Very useful when storage latency is high and/or bandwidth is low
- With NVM, nothing much to hide ┓(̄Д ̄)┏
- But how about in a distributed and/or replicated file system?
  - Anything else interesting?

Distributed storage on NVM

- Octopus[Lu et al., ATC'17] achieved low latency zero-copy RDMA transmission on server side
  - Server NIC use DMA to read data directly from FS image
  - They reported ≥85% bandwidth usage and ~6µs end-to-end latency. Can any application benefit from asynchronous I/O? If so, are existing APIs good enough?

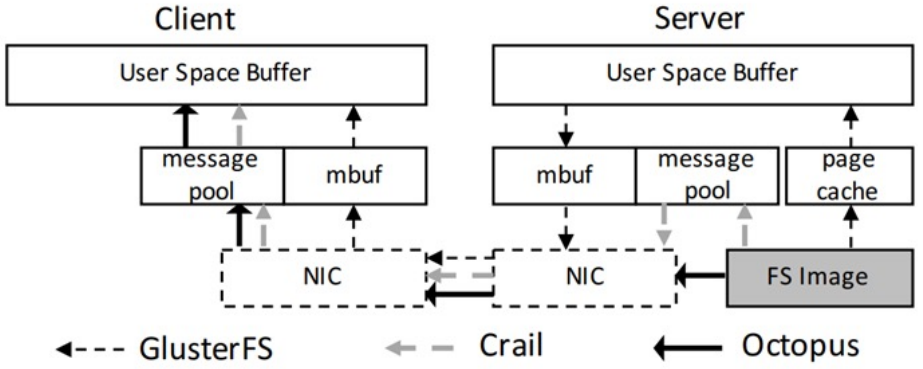


Figure 4: Data Copies in a Remote I/O Request

- PASTE[Honda et al., NSDI'18] allocates message pool directly in NVM, achieving zero-copy networking interface on client side
  - Can/should we combine NVM-based message pool into an FS? Any applications that would benefit from it?

User-space FS and storage

- FUSE
  - User-space *extension* of VFS from kernel
  - For FS functionalities not worth putting in kernel
- SPDK
  - Pure user-space storage stack (block device driver, FS-like block device abstraction, etc.)
  - Performs better: actively polling for I/O completion; lockless I/O queues; no syscalls
  - Implemented as a C/C++ library

User-space FS and storage on NVM

- ZoFS[Dong et al., SOSP'19] relaxes file protection from every single file to a group of files (coffer) with the same access privilege
  - Reduces system call from every file operation to only the first access to each coffer
  - Can we also relax the protection of (a part of) page table to achieve page remapping in user space?
    - If we need kernel support for TLB shootdown, can we do it lazily?
    - More hardware design proposals? (SPOT[Wang et al., ISCA'18])
- EvFS[Yoshimura et al., HotStorage'19] is purely in user space, inspired by SPDK
  - Designed to live in a single protection domain

Some mixture of all

- SplitFS[Kadekodi et al., SOSP'19] serves read and write operations in user space.
  - It DAX-mmaps pages into user space and translates read() and write() into load and store instructions. Metadata accesses are done in kernel
  - Provides relink() function that links pages in one file to another by modifying only page mapping metadata
  - Writes are staged in a new file and inserted into files with relink()
- Libnvmmio[Choi et al., ATC'20]
  - takes a similar approach to translate read() and write() into load and store, but uses per-block logging and epoch-based checkpointing for persistence

Some mixture of all

- Assise[Anderson et al., OSDI'20] uses NVM-based client-local caches to lower the latency of distributed FS
  - Each node of the cluster maintains synchronized write logs and data replicas of different hotness on various media like NVM, SSD and remote storage
  - Write cache on NVM takes the form of per-process logs with variable-sized entries, replicated across the cluster
  - Read cache is kept in DRAM
  - When sharing FS states among processes, it implements a lease-based crash consistent cache coherence protocol to provide prefix crash consistency (similar to buffered durable linearizability)

Observations

- State-of-art systems make interesting trade-offs between the amount of process-private (DAX-mmap()ed) NVM per system call and the range of protection to serve their needs
  - Page, file, write cache, coffer, or the whole system
- Reducing copying is a common effort among recent proposals, which is also a clever way of using NVM's byte-addressability. However, for applications with frequent reads/updates to staging data for I/O, DRAM's even lower latency may outweigh the data copy overhead between DRAM and NVM
  - e.g. Assise's read cache in DRAM