Sub-Zero: Zero-copy IO for Persistent Main Memory File Systems

Juno Kim, Yun Joon Soh, Joe Izraelevitz*, Jishen Zhao, Steven Swanson

UC San Diego, University of Colorado, Boulder*

Non-Volatile Systems Laboratory

Department of Computer Science & Engineering

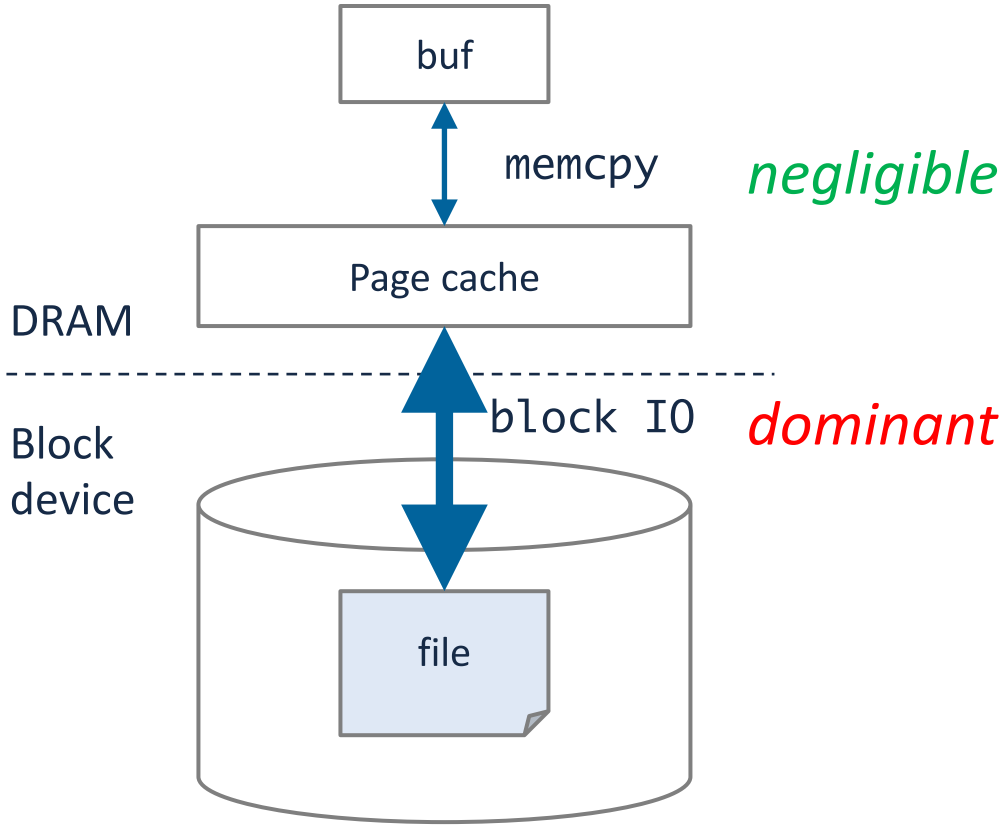
University of California, San Diego

NVSL

UCSDCSE Computer Science and Engineering

Copy-based conventional file IO interface

- read(), write() system calls rely on copy-based semantics
  - User provides the buffer address
  - Data is copied between the buffer and the storage media via page cache
- The first movement “memcpy” is not significant when storage is slow



NVSL

What if storage is fast enough?

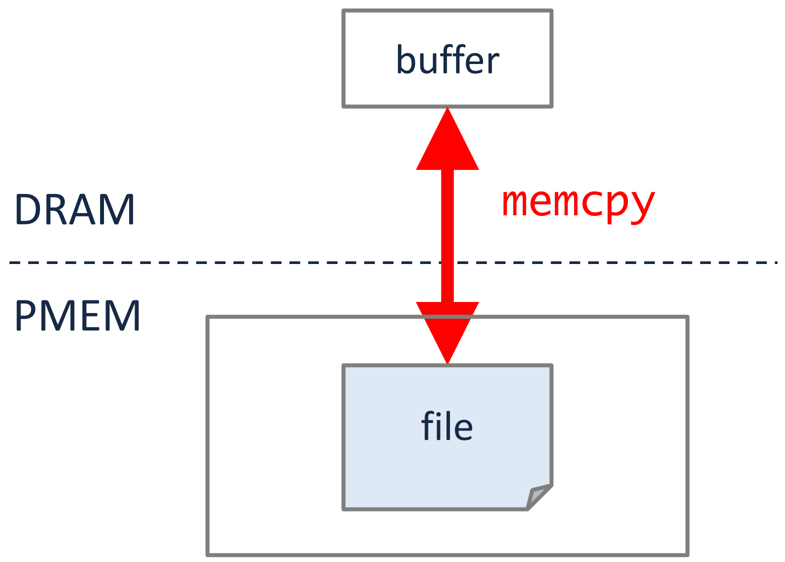
- **Persistent memory (PMEM)**: new storage with near-DRAM speed
  - Orders of magnitude faster than disks/SSDs
  - Only 2~3x slower than DRAM
- PM allows direct access (DAX)
  - File systems bypass the page cache
- DAX file systems
  - Ext4, XFS in DAX mode (Linux)
  - NOVA (UCSD), Strata, SplitFS (UT Austin)
  - And more

Technology	Latency	
	Read	Write
DRAM	0.1 µs	0.1 µs
Persistent Memory	**0.3 µs**	**0.1 µs**
NVMe SSD	120 µs	30 µs
SATA SSD	80 µs	85 µs
HDD	10 ms	10ms

Conventional file IO on PMEM

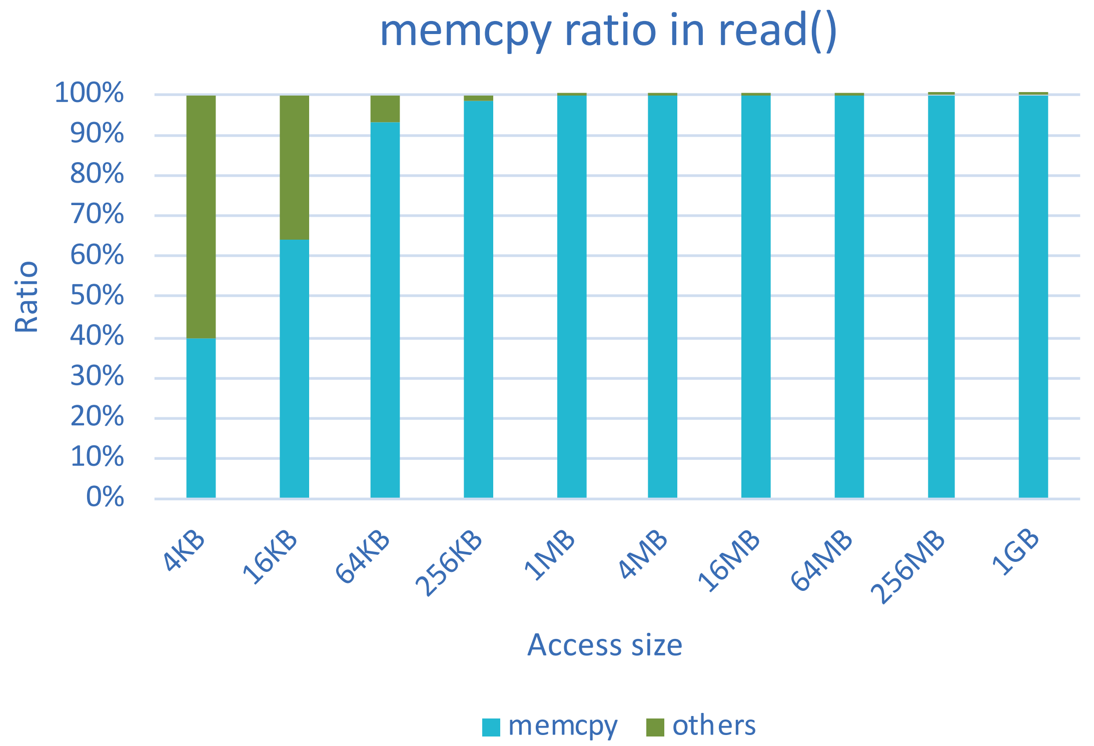
- Page cache is bypassed by DAX
  → Direct memory copying between user buffer and PM

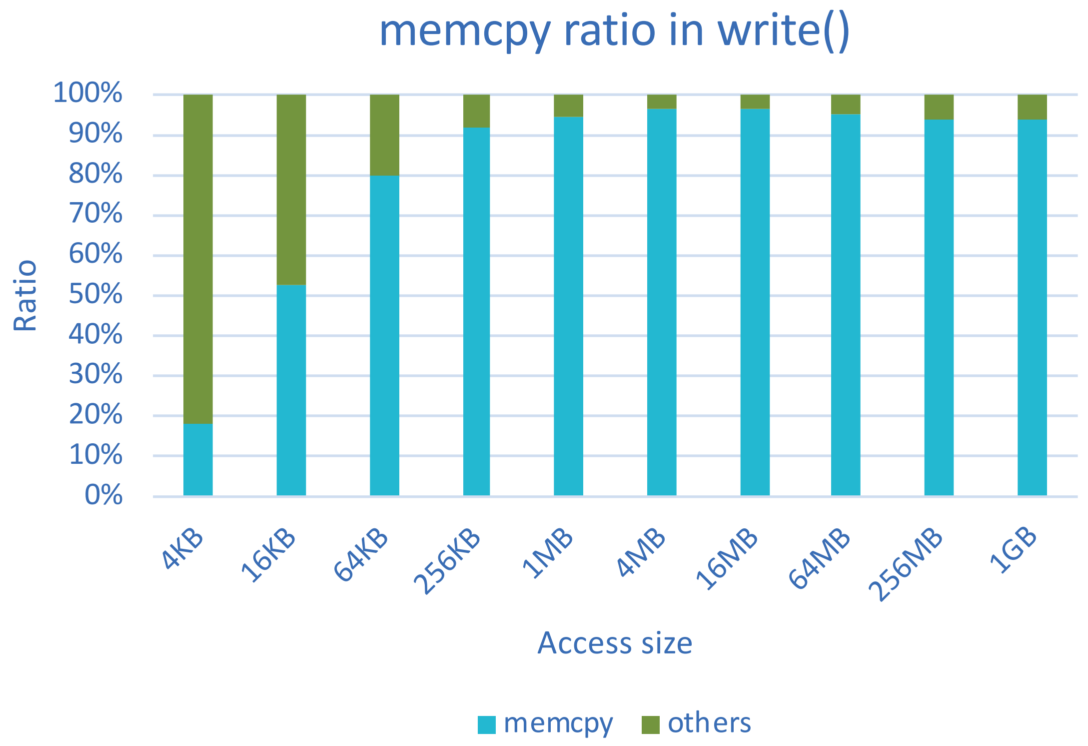
- The last movement is enforced by the read(), write() interface



Copying dominates as access size grows

** Measured on NOVA file system[1]*



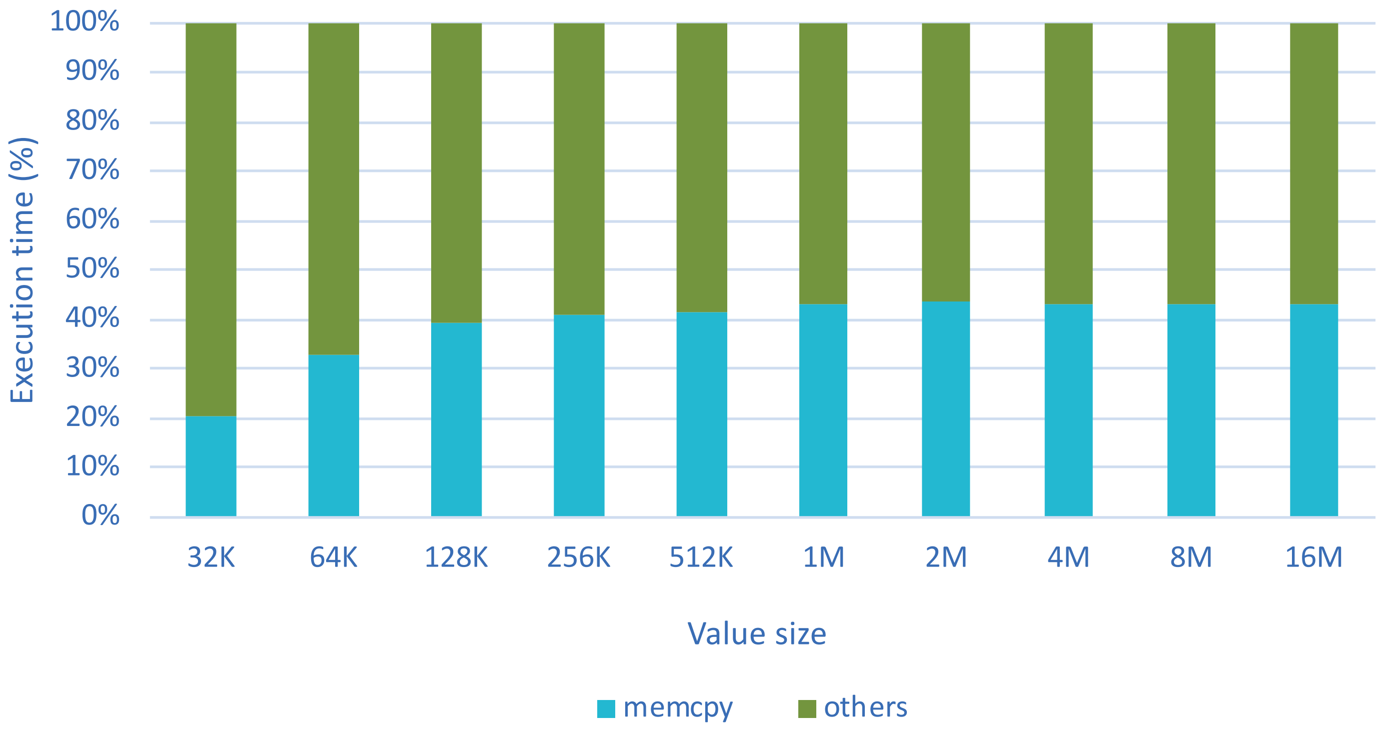


[1] NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories, Jian Xu, Steven Swanson, FAST’16

NVSL

Copying overhead in real application

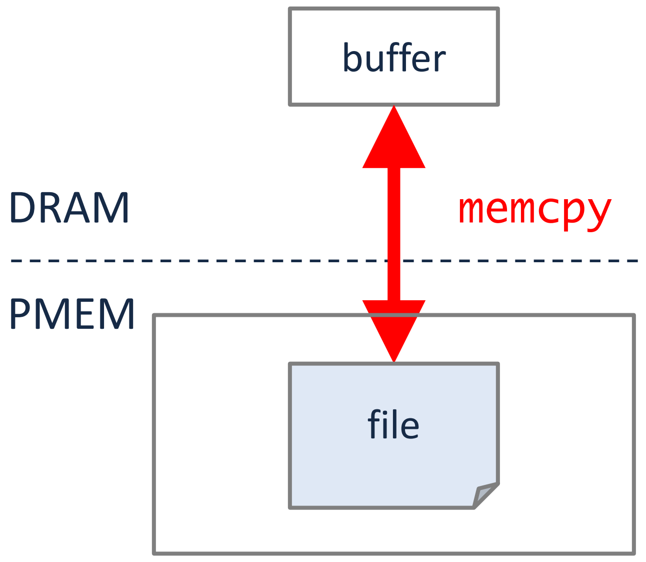
- Kyoto Cabinet: high-performance key-value library
- <u>Memcpy in write() takes 20~45% of SET operation</u>



NVSL

How can we remove this memcpy?

- New IO interface is necessary
  - Copying is the property of read(), write() semantics
  - New interface must allow direct access to remove copying
- Isn't mmap() enough for this purpose?



mmap() complicates programming

- Lack of atomicity
  - Atomic unit of update is only 8-byte by processor
  - Failure-recovery can yield inconsistent states
- Lack of concurrency control
  - Concurrent access might observe partial data

Programmers must implement necessary mechanisms on their own.

Sub-Zero IO

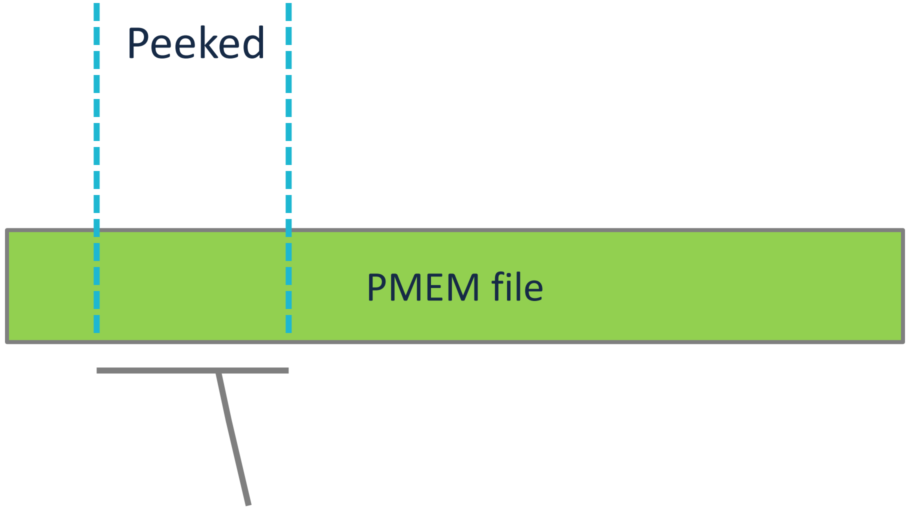
- New system calls that access PMEM files *without copy-based semantics*
  - Sub-Zero preserves the ease of use that read(), write() provide
  - Sub-Zero provides high-performance similar to mmap()
- Two key primitives: **peek(), patch()**

Rest of the talk

- Sub-Zero IO overview
  - Peek()
  - Patch()
- Implementation
- Performance evaluation
- Conclusion

peek() system call

- Returns a pointer to a PMEM region
  - The pointer is equivalent to a *snapshot* of the file contents
  - The pointer is *immutable*
- Allows easier programming than mmap(), because
  - Peek() works *at any arbitrary offset*
  - Peek() captures a private snapshot *atomically*
- Unpeek() closes the mapping opened by peek()



- Read-only
- Isolated from other file modifications

peek() example 1: basic

```
// peek the first 4KB of a PMEM file
int fd = open("foo", O_RDONLY);              // Open the target file
char *buf = peek(fd, 0, 4096);               // Peek its contents
printf(“%s\n”, buf);                         // Print the contents
unpeek(buf);                                 // Unpeek the contents
```

NVSL

peek() example 2: immutability

```
// peek the first 4KB of a PMEM file
int fd = open("foo", O_RDONLY);             // Open the target file
char *buf = peek(fd, 0, 4096);              // Peek its contents
printf(“%s\n”, buf);                        // Print the contents
*buf = ‘a’;                                 // Segmentation fault!
unpeek(buf);                                // Unpeek the contents
```

NVSL

peek() example 3: isolation

```
// Thread 1: peek the first 4KB of a PMEM file
int fd = open("foo", O_RDONLY);
char *buf = peek(fd, 0, 4096);
...
...
printf("%s\n", buf);      // print original contents!
...
unpeek(buf);
close(fd);
```

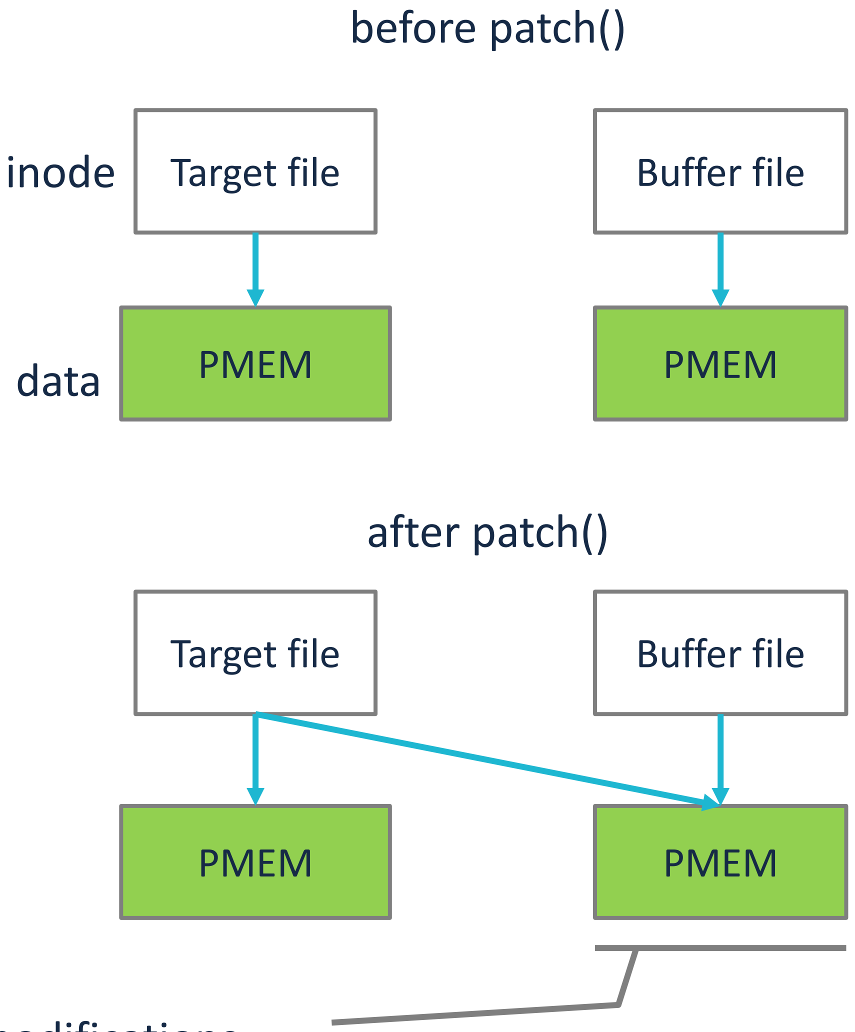
```
// Thread 2: update the peek()'ed region
// of the same file
int fd = open("foo", O_WRONLY);
char *buf = malloc(4096);
memset(buf, 0xab, 4096);
write(fd, buf, 4096);     // copy-on-write to
...                       // a new 4KB
free(buf)
close(fd);
```

NVSL

patch() system call

- Modifies a file by *merging* the contents of a buffer into the file
  - The buffer *becomes* parts of the file
  - The buffer is *immutable* after patch()

- The buffer must be in PMEM



- Read-only
- Isolated from other file modifications

patch() example 1: basic

```
// Update the first 4KB of a PMEM file
int fd = open(”/mnt/foo", O_RDONLY);                            // Open the target file
int pool_id = create_pmem_pool(“/mnt”, 1073741284);             // Create a pool
void *buf = alloc_pmem(pool_id, 0, 4096);                       // Allocate a PMEM buffer
memset(buf, ‘\0’, 4096);                                        // Populate new data in the buffer
patch(fd, buf, 4096, 0);                                        // Patch it into the file
free_pmem(buf);                                                 // Unmap the buffer
```

NVSL

patch() example 2: immutability

```
// Update the first 4KB of a PMEM file
int fd = open("foo", O_RDONLY);                        // Open the target file
int pool_id = create_pmem_pool(“/mnt”, 1073741284);    // Create a pool
void *buf = alloc_pmem(pool_id, 0, 4096);              // Allocate a PMEM buffer
memset(buf, ‘\0’, 4096);                               // Populate new data in the buffer
patch(fd, buf, 4096, 0);                               // Patch it into the file
*(char*)buf = ‘a’;                                     // Segmentation fault!
free_pmem(buf);                                        // Unmap the buffer
```

NVSL

Implementation

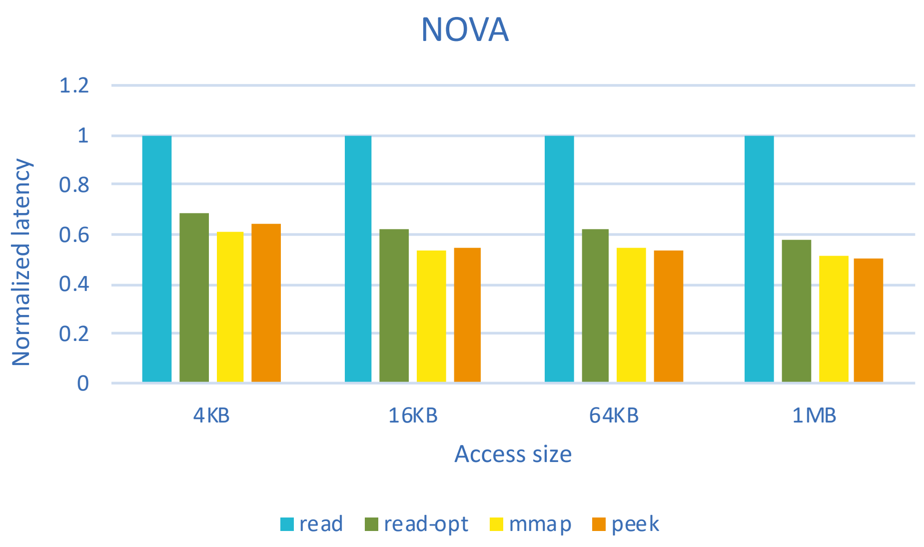
- Implemented Sub-Zero IO in NOVA and XFS-DAX
  - Under Linux kernel 4.19
- SubZero can be implemented without invasive changes if the file system
  - Allows multiple files to share data pages
  - Supports COW data update when a write updates shared pages
- Both file systems support these features
  - NOVA supports COW for strong data consistency
  - XFS-DAX supports page sharing/COW for "reflink"

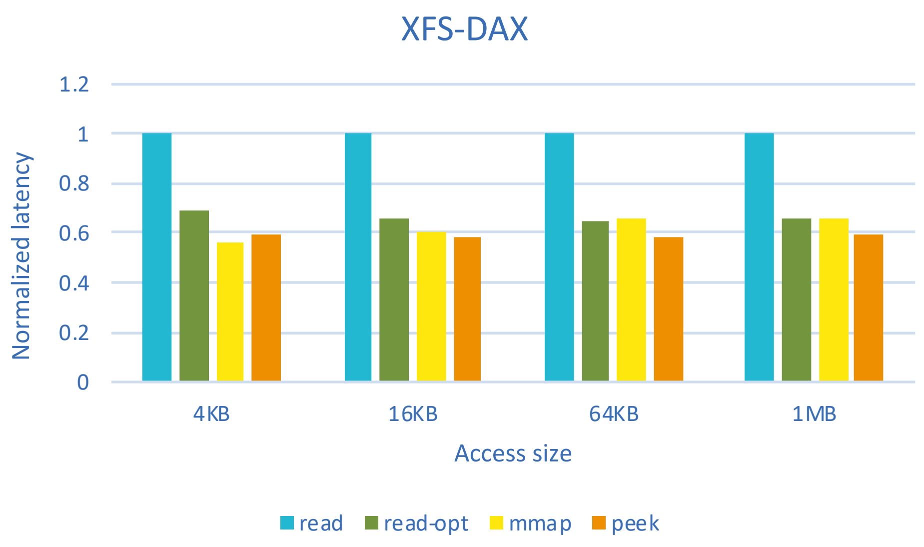
Performance Evaluation

- Micro-benchmark
  - Basic performance compared to read(), write(), and mmap()
  - Latency includes the time to allocate/populate/free the buffer
- Application
  - Apache Web Server: widely-deployed web server
  - Kyoto Cabinet: high-performance key-value store library
- Measured on Intel’s Optane DC Persistent Memory

Performance of peek()

- peek() performs up to 2x faster than read() depending on the reuse of buffer
- peek() performs almost similar to mmap()

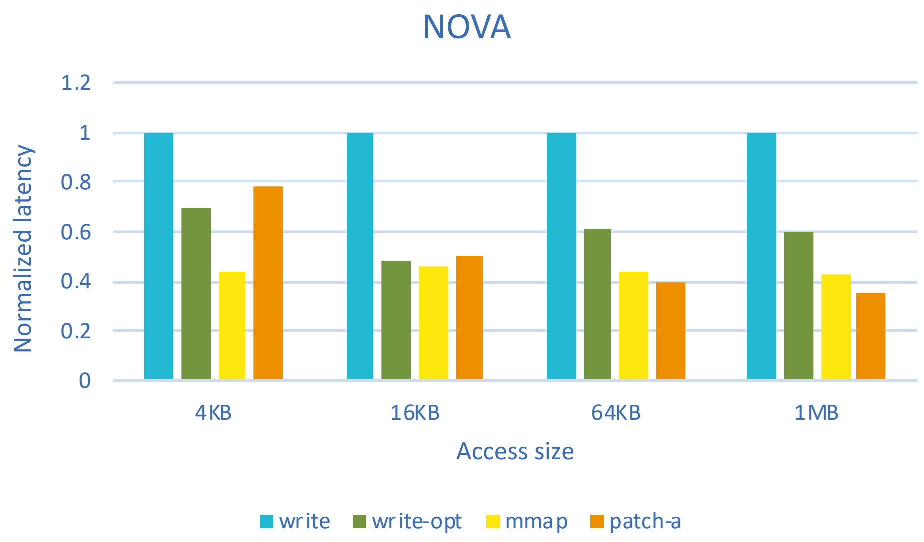


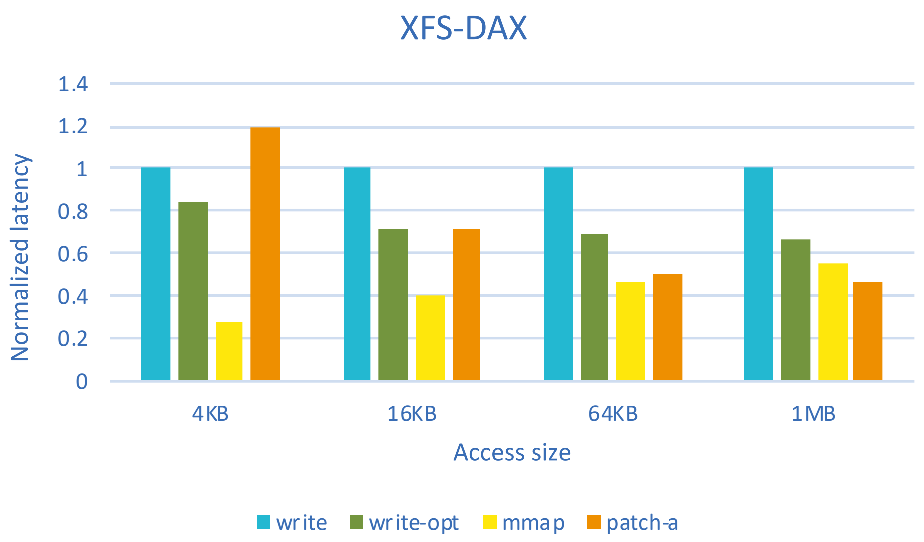


NVSL

Performance of patch()

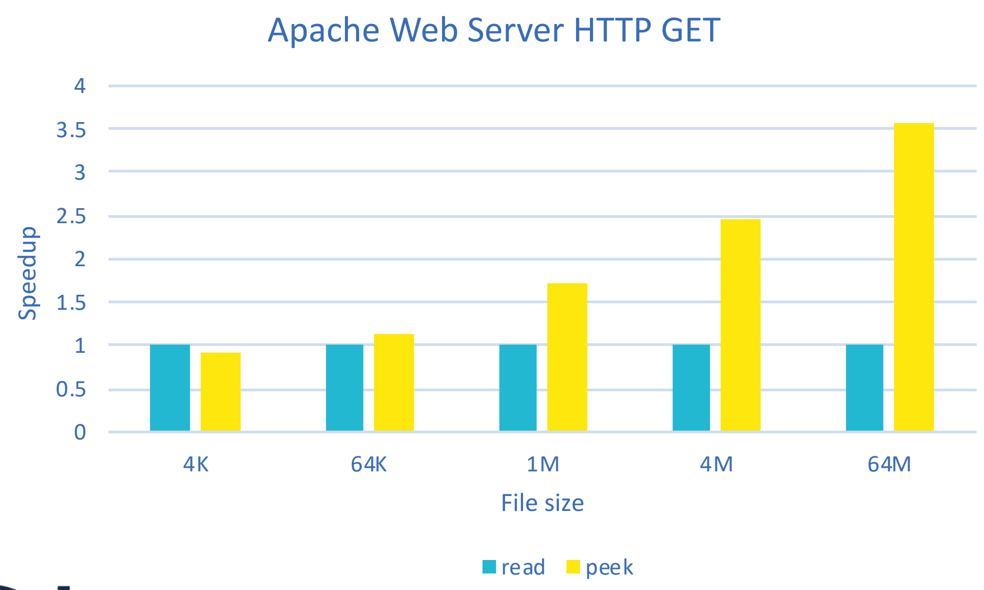
- patch() outperforms write() as the access size grows
- The speedup is up to 2.8x and 2.2x in NOVA and XFS-DAX, respectively
- patch() is slower than mmap() under 64kB, but becomes faster beyond 64kB

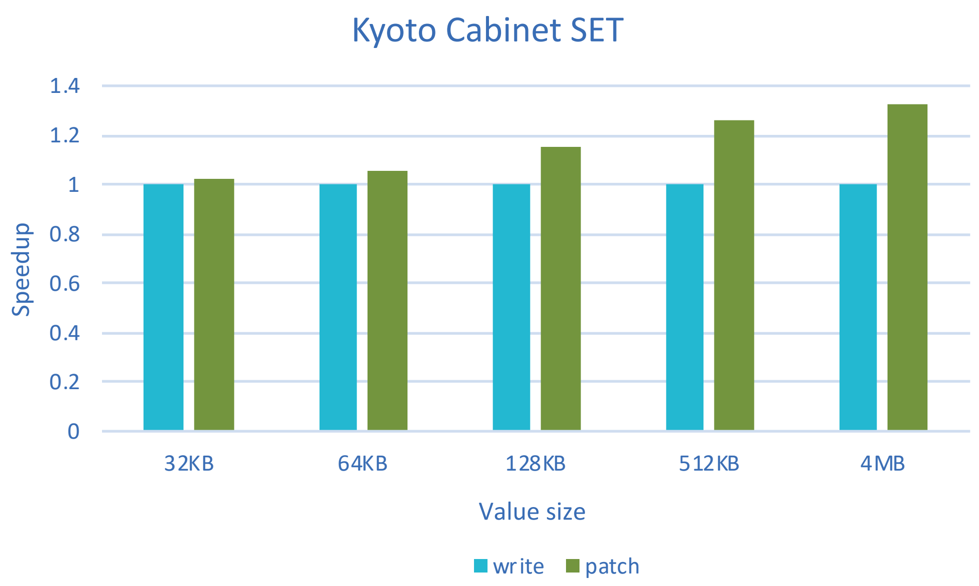




Application performance

- Apache Web Server: HTTP GET with peek() → **3.6x**
- Kyoto Cabinet: SET with patch() → **1.3x**





NVSL

Conclusion

- New IO system calls that offer high performance on PMEM file systems
  - Simple API: peek(), patch()
  - Low overhead: no data copying
- Easier programming than mmap()
  - Provides atomicity, isolation in kernel
- Require minimal changes to applications