

Persistent memory hardware and programming

Adrian Jackson

a.jackson@epcc.ed.ac.uk

[@adrianjhpc](https://twitter.com/adrianjhpc)

Persistent/Non-volatile memory

- Persistent/Non-volatile memory stores data after power off
 - SSDs (NAND Flash) are common examples
 - Similar technology in memory cards for your phones, cameras, etc...
- These store data persistently but are generally slow and less durable than volatile memory technologies (i.e. DRAM memory)
- Traditional non-volatile technology is accessed through a block device interface
 - Chunks of data at a time (i.e. 4kb), asynchronous access

NVDIMMs

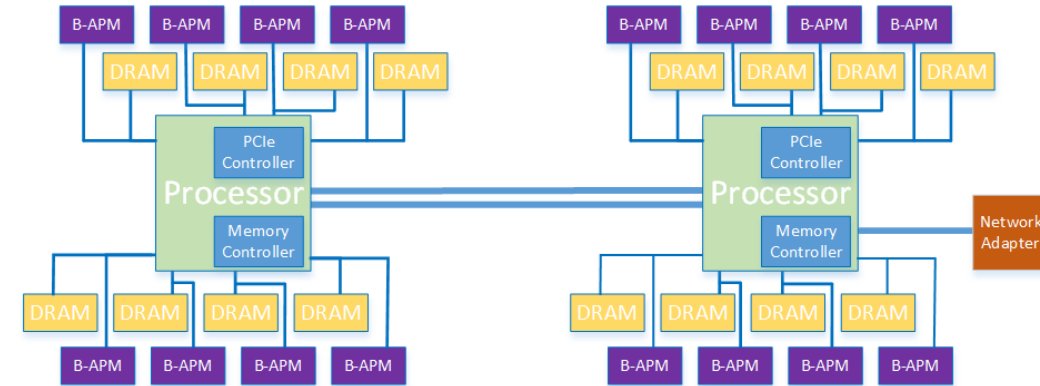
- JEDEC standard on non-volatile memory DIMMs
- NVDIMM-F
 - Traditional flash solution with controller on board
 - NAND flash performance and size
- NVDIMM-N
 - DRAM with Flash for backup
 - Separate power supply (i.e. super capacitors) allow data to be copied to flash on power failure
 - Limited by DRAM size and capacitor sizes
 - DRAM performance and size
- NVDIMM-P
 - Channel support for mixed memory types
 - Protocol to enable transactional access
 - Different access latencies allowed between media types
 - Intel Optane DCPMM, technically, does not implement the NVDIMM-P standard, but it is conceptually NVDIMM-P
 - What I call Byte-Addressable Persistent Memory (B-APM)

Intel Optane DCPMM

- Persistent/Non-volatile RAM/B-APM
 - Optane memory
- Much larger capacity than DRAM
 - Hosted in the DRAM slots, controlled by a standard memory controller
- Slower than DRAM by a small factor, but significantly faster than SSDs
- Read/write asymmetry and other interesting performance factors
- Cache coherent data accesses
 - Byte addressable (cache line)
- High endurance (5 year warranty)

Placement

- As B-APM is in-node, placement is important
 - Configuration can be variable
 - Currently need one DRAM per memory channel
 - Can match DRAM-B-APM or have more B-APM
- Capacity (both DRAM and B-APM) affected
- Memory controller deals with mixed configuration
 - Variable latency on memory DIMMs
 - Requires asynchronous or non-blocking memory operations
- Optane DIMMs can be striped on non-striped
 - One memory area per DIMM, or one memory area per socket, striped across DIMMs



Optane DCPMM – NUMA issues

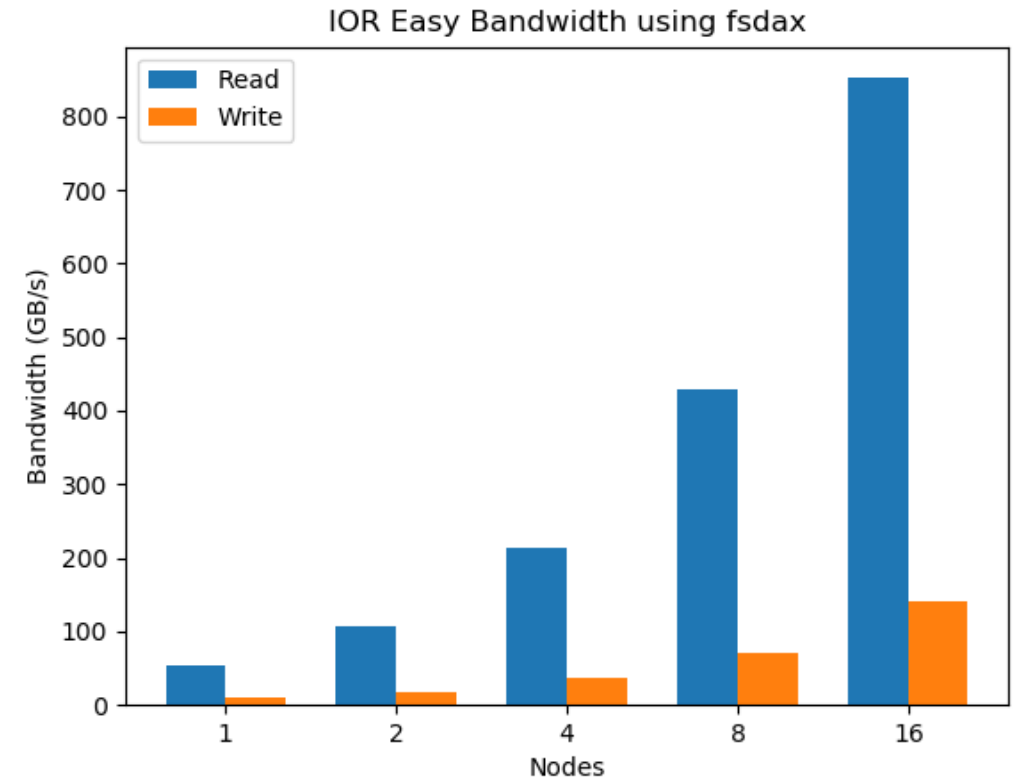
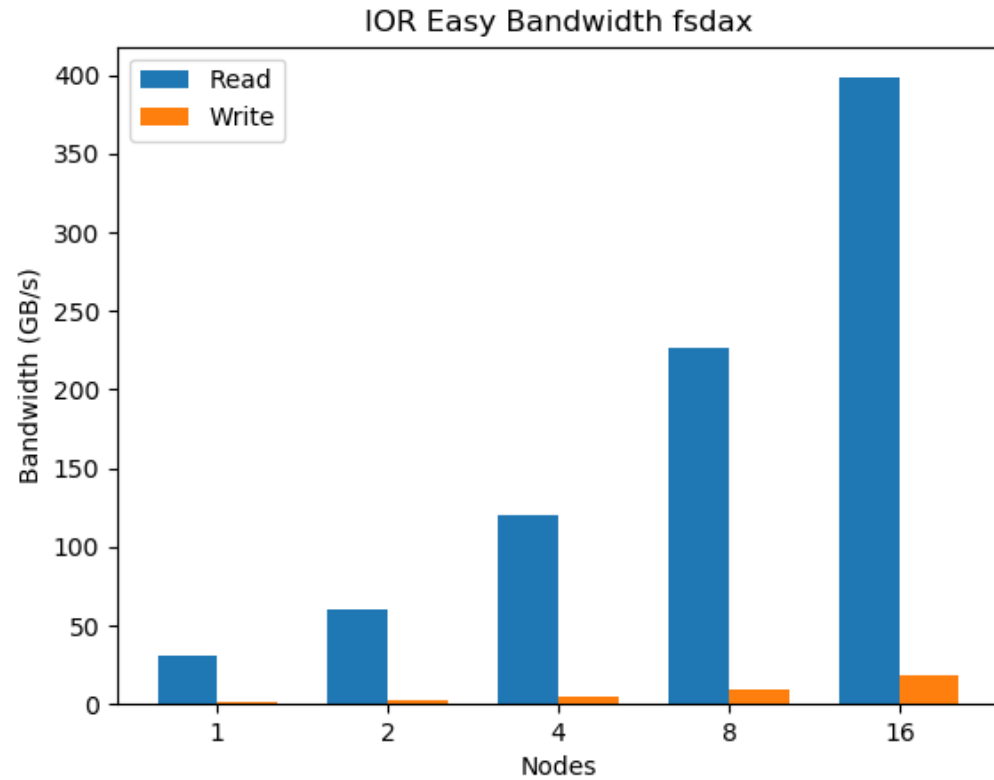
- Socket based systems means NUMA when not a single socket system
 - Performance dependent on using local memory
- Factor ~4x write performance for using local memory when fully populating nodes
- Factor ~2x read performance for using local memory when fully populating nodes
- Getting NUMA information in an application
 - Intel specific:

```
unsigned long GetProcessorAndCore(int *chip, int *core){  
    unsigned long a,d,c;  
    __asm__ volatile("rdtscp" : "=a" (a), "=d" (d), "=c" (c));  
    *chip = (c & 0xFFF000)>>12;  
    *core = c & 0xFFF;  
    return ((unsigned long)a) | (((unsigned long)d) << 32);  
}
```

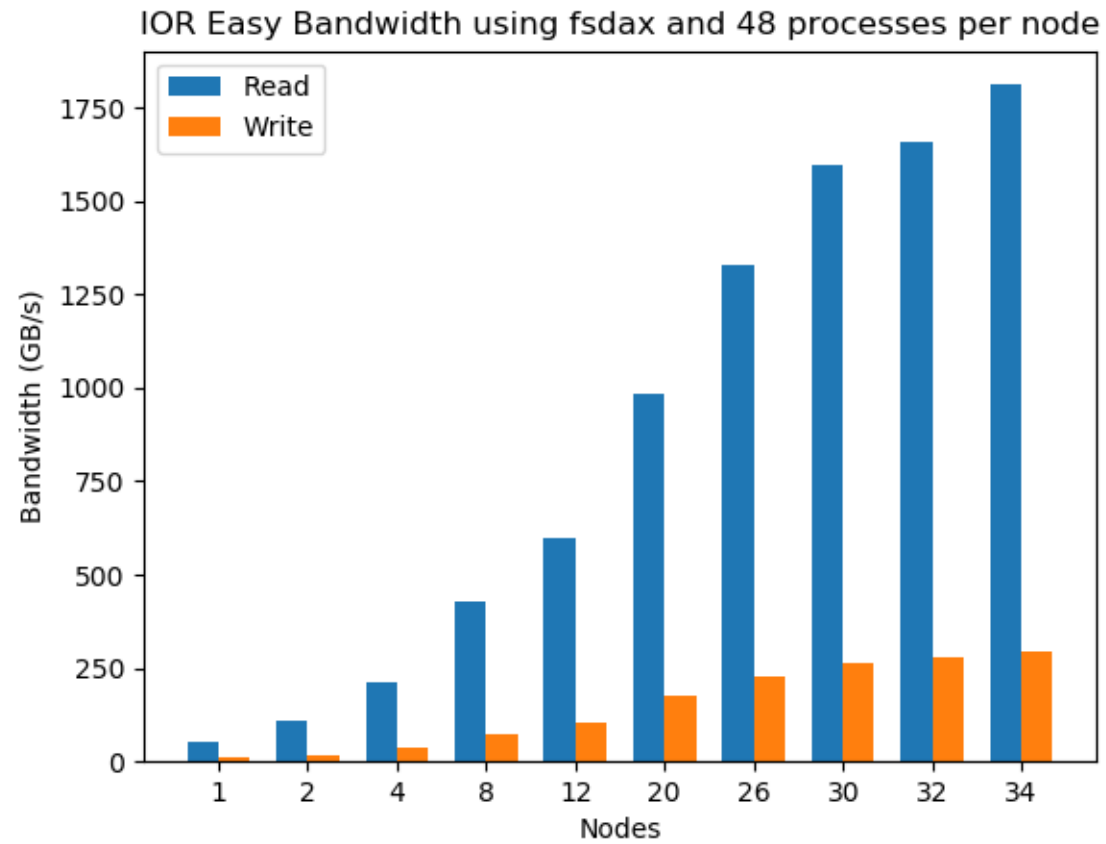
- Processor agnostic

```
unsigned long GetProcessorAndCore(int *chip, int *core){  
    return syscall(SYS_getcpu, core, chip, NULL);  
}
```

Optane NUMA performance



Optane performance



Optane Performance asymmetry

- Read is ~3x slower than DRAM
- Write is ~7x slower than DRAM
- Read is 4x-5x faster than write for Optane
- Write queue issues can mean variable performance
 - Optane has active write management
 - On-DIMM controller
- Accesses are coalesced into blocks of i.e. 256 bytes

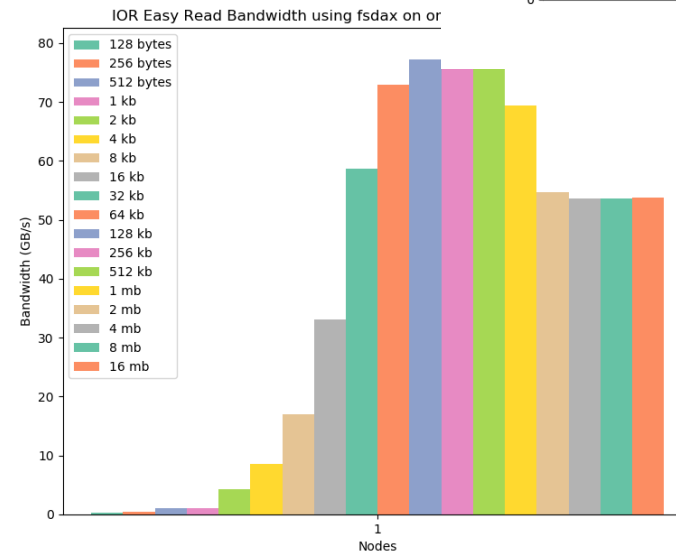
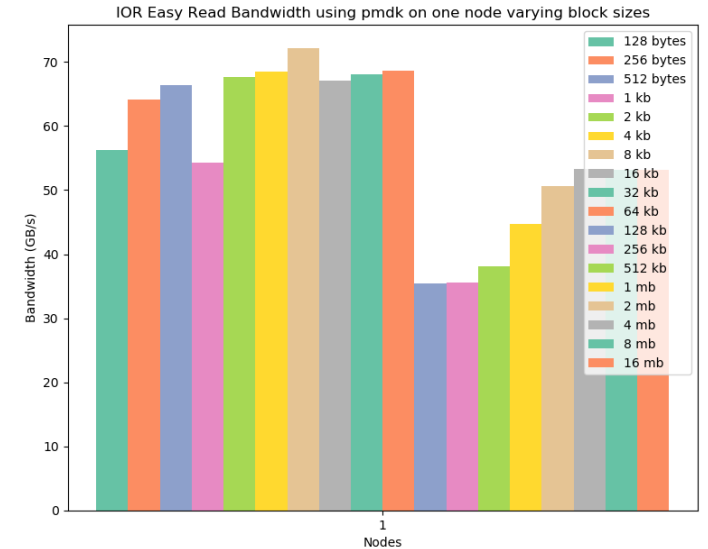
INTEL® OPTANE™ DC PERSISTENT MEMORY PERFORMANCE DETAILS

- Intel® Optane™ DC persistent memory is programmable for different power limits for power/performance optimization
 - 12W – 18W, in 0.25 watt granularity - for example: 12.25W, 14.75W, 18W
 - Higher power settings give best performance
- Performance varies based on traffic pattern
 - Contiguous 4 cacheline (256B) granularity vs. single random cacheline (64B) granularity
 - Read vs. writes
 - Examples:

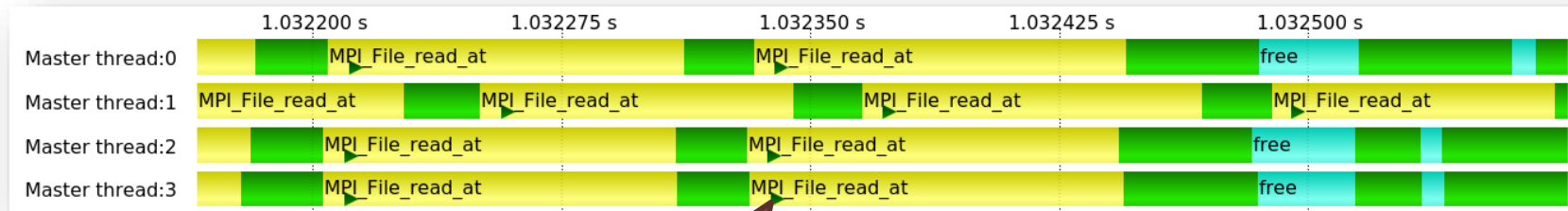
Granularity	Traffic	Module	Bandwidth
256B (4x64B)	Read	256GB, 18W	8.3 GB/s
256B (4x64B)	Write		3.0 GB/s
256B (4x64B)	2 Read/1 Write		5.4 GB/s
64B	Read		2.13 GB/s
64B	Write		0.73 GB/s
64B	2 Read/1 Write		1.35 GB/s

Byte Addressable/Granularity

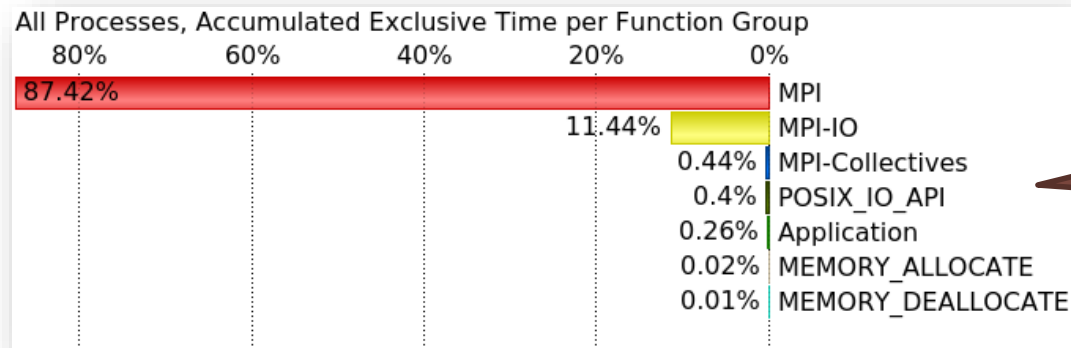
- A key feature of B-APM
 - Data access is the same cost for any size of operation
 - Not really, but much closer than for standard files
- Byte-addressable just like standard memory
 - Individual bytes accessible without large operations
 - In reality, cache-line level access
- I/O and data operations can be small
 - Restructure I/O/applications



Granularity



Individual I/O Operation



I/O Runtime Contribution

Standard I/O Programming

```
my_file = open("data_out.dat", O_RDWR);  
read(my_file, a, bufsize);  
write(my_file, b, bufsize);  
close(my_file);
```

```
open(fid, "checkpoint.dat")  
do n = 0, 2*nharms  
    write(fid) ((q(i, j, k, n), i=-1, imax1), j=-1, jmax1), k= 1, kmax1)  
end do  
close(fid)
```

Standard I/O Programming

- Writing to O/S buffer, operating system writes that back to the file
 - Potential for O/S caching
 - Writes data in large chunks, bad for random access
 - Requires interaction with O/S
 - I/O consistency application responsibility
 - Flush required to ensure actual persistency
- Required because of the nature of previous I/O devices
 - Asynchronous
 - I/O controller
 - Shared

Parallel I/O Programming

- Library function to write to shared or separate files
- Library collects data and orchestrates writing to the actual file
- Still block based ultimately, requires parallel filesystem to enable multiple processes writing at once to the same file
 - Or multiple processes accessing multiple files at once

```
call mpi_file_write(fid, linelength, 1, MPI_INTEGER, MPI...
```

Optimising I/O

- Optimising I/O performance can be done in a number of ways
 - Ensuring the minimal number of actual block writes is done
 - Ensure the minimal number of filesystem operations are performed
 - Use faster I/O hardware
 - Use multiple bits of I/O hardware
 - Map the file to memory for I/O operations
- Optimising hard for a range of use cases
 - Small I/O operations
 - Non-contiguous I/O operations
 - Contention on shared resources (network or filesystem)

Memory-mapped files

```
my_file = open("data_file.dat", O_RDWR);  
data = mmap(NULL, filesize, PROT_READ|PROT_WRITE, MAP_SHARED, my_file, 0);  
close(fd);  
data[0] = 5.3;  
data[1] = 75.4;
```

- Memory-mapping a file copies the file into main memory
 - Requires sufficient main memory to contain the file
- Operations can then be undertaken on that data in standard memory format
 - Load/store, cache-line level accesses
- Persistence to file requires flush
 - msync
 - Done on page level
 - Every dirty page written back to file system
- Page cache supports dirty flag
 - Only dirty pages written back
- Volatile until persisted

PMDK

- To utilise persistent memory functions are need to allocated, deallocate, and persist memory in the address ranges of the NVDIMMs
- mmap approach is adopted
 - Except the persistent memory data is not mapped into DRAM
 - And there is no initial read of data because it is already in persistent memory
 - And persisting is done on cache lines
- Functions to help run on systems with and without persistent memory
- PMDK has various different approaches for using persistent memory
 - <https://github.com/pmem/pmdk>

pmem

- pmem lowest level approach available in PMDK
- Functions to memory map persistent memory and persist data to the hardware
- Functions to do optimised memcpy, memmove etc...
- User's responsibility to ensure data is safely persisted

pmem example

```
double *a, *b, *c;
pmemaddr = pmem_map_file(path, array_length, PMEM_FILE_CREATE|PMEM_FILE_EXCL,
                        0666, &mapped_len, &is_pmem)

a = pmemaddr;
b = pmemaddr + (*array_size+OFFSET)*BytesPerWord;
c = pmemaddr + (*array_size+OFFSET)*BytesPerWord*2;

#pragma omp parallel for
for (j=0; j<*array_size; j++){
    a[j] = b[j]+scalar*c[j];
}

pmem_persist(a, *array_size*BytesPerWord);
pmem_unmap(pmemaddr, mapped_len);
remove(path);
```

pmem_map_file

```
void *pmem_map_file(const char *path, size_t len, int flags, mode_t mode, size_t *mapped_lenp, int *is_pmemp);
```

- Path is the location of the pmem mount
 - /mnt/pmem_fsdax0
 - /dev/dax0.0
- Flags
 - **PMEM_FILE_CREATE** - Create the file named *path* if it does not exist. *len* must be non-zero and specifies the size of the file to be created. If the file already exists, it will be extended or truncated to *len*.
 - **PMEM_FILE_EXCL** - If specified in conjunction with **PMEM_FILE_CREATE**, and *path* already exists, then **pmem_map_file()** will fail with **EEXIST** (fsdax only)
 - **PMEM_FILE_TMPFILE** - Create a mapping for an unnamed temporary file (fsdax only)
- For devdax *len* must be equal to either 0 or the exact size of the device

pmem_persist

```
void pmem_persist(const void *addr, size_t len);
```

- Force the data at address `addr` to `addr+len` to be written to the persistent media.
- If this is on B-APM it will write directly back to the hardware (no O/S calls)
- It may write slightly more back (cache line size) than specified in `len`
- Before this call data *may* be persisted, but no guarantee

```
int pmem_msync(const void *addr, size_t len);
```

- Works on non-B-APM as well

```
Void pmem_persist(const void *addr, size_t len){
```

```
/* flush the processor caches */
```

```
    pmem_flush(addr, len);
```

```
/* wait for any pmem stores to drain from HW buffers */
```

```
    pmem_drain();
```

```
}
```

Memory operations

```
void *pmem_memmove_persist(void *pmemdest, const void *src,  
size_t len);
```

```
void *pmem_memcpy_persist(void *pmemdest, const void *src,  
size_t len);
```

```
void *pmem_memset_persist(void *pmemdest, int c, size_t len);
```

- Copy, move, and set functions that automatically persist the data as well
- For control of persistence using the standard functions instead

```
void * pmem_memmove_persist(void *pmemdest, const void *src, size_t  
len) {
```

```
    void *retval = memmove(pmemdest, src, len);
```

```
    pmem_persist(pmemdest, len);
```

```
    return retval;
```

```
}
```

pmem2

- Recognised power failure domains at the software level
 - Processor
 - Memory Controller
 - Nothing
- Enables configuration of persistent granularity for data structures
 - PMEM2_GRANULARITY_BYTE
 - PMEM2_GRANULARITY_CACHE_LINE
 - PMEM2_GRANULARITY_PAGE
- Provides optimised memory operations to work with these different configurations
 - pmem2_get_persist_fn
 - pmem2_get_flush_fn, pmem2_get_drain_fn
 - pmem2_get_memcpy_fn, pmem2_get_memmove_fn

```
int fd;
struct pmem2_config *cfg;
struct pmem2_map *map;
struct pmem2_source *src;
pmem2_persist_fn persist;

fd = open(argv[1], O_RDWR);
pmem2_source_from_fd(&src, fd);
pmem2_config_set_required_store_granularity
(cfg, PMEM2_CACHE_LINE);
pmem2_map_new(&map, cfg, src);

char *addr = pmem2_map_get_address(map);
size_t size = pmem2_map_get_size(map);

strcpy(addr, "hello, persistent memory");

persist = pmem2_get_persist_fn(map);
persist(addr, size);

pmem2_unmap(&map);
pmem2_source_delete(&src);
pmem2_config_delete(&cfg);
```

libpmem(2)

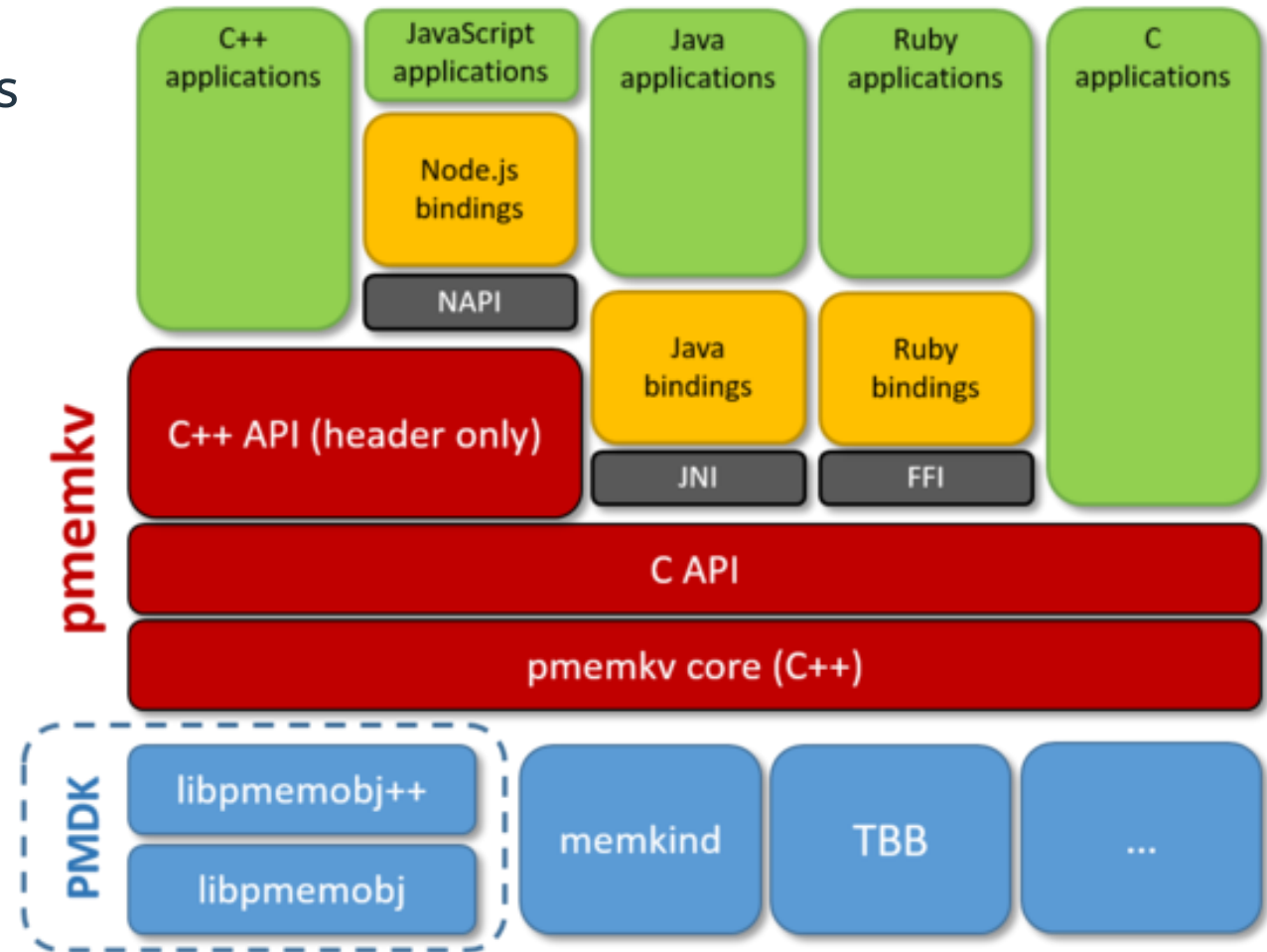
- Basic persistence functionality provided by the PMDK pmem library
 - Required if you want to use the persistent feature of persistent memory
- If large memory is all that is required libmemkind is available
- pmem defers responsibility to the programmer to ensure persistence is properly handled
 - Separates out memory creation and persistent calls
 - Enables control over persistent granularity
 - Enables maximum performance
 - But also maximum danger

PMDK higher level options

- libpmemobj
 - Transactional object store
 - Providing memory allocation, transaction
- Pools
 - libpmemblk
 - Blocks, all the same size, that are atomically updated
 - libpmemlog
 - Log file on persistent memory
- pmemkv
 - Key value store

pmemkv

- Storage engine with simple operations
 - Start
 - Stop
 - Put
 - Get
 - Remove
 - Exists
- Also further iterator operations
 - Count
 - All
 - Each



pmemkv

```
int main() {  
    KVEngine* kv = KVEngine::Start("vsmmap", "{\\"path\\":\\"/dev/shm/\\"}");  
    KVStatus s = kv->Put("key1", "value1");  
    string value;  
    s = kv->Get("key1", &value);  
    s = kv->Remove("key1");  
    delete kv;  
    return 0;  
}
```

pmemkv

Storage Engines

`pmemkv` provides multiple storage engines that conform to the same common API, so every engine can be used with all language bindings and utilities. Engines are loaded by name at runtime.

Engine Name	Description	Experimental?	Concurrent?	Sorted?
blackhole	Accepts everything, returns nothing	No	Yes	No
cmap	Concurrent hash map	No	Yes	No
vsmmap	Volatile sorted hash map	No	No	Yes
vcmap	Volatile concurrent hash map	No	Yes	No
tree3	Persistent B+ tree	Yes	No	No
stree	Sorted persistent B+ tree	Yes	No	Yes
caching	Caching for remote Memcached or Redis server	Yes	No	-

Contributing a new engine is easy and encouraged!

libpmemobj

- Object store with transactions
- Provides functions to create and manage data in persistent memory
- Provides macros and functions to add and remove data from object store
- Node local

```
PMEMobjpool *pmemobj_open(const char *path, const char *layout);  
void pmemobj_close(PMEMobjpool *pop);  
PMEMoid pmemobj_root(PMEMobjpool *pop, size_t size);  
int pmemobj_tx_add_range(PMEMoid oid, uint64_t off, size_t size);  
int pmemobj_tx_add_range_direct(const void *ptr, size_t size);  
PMEMoid pmemobj_tx_alloc(size_t size, uint64_t type_num);  
int pmemobj_tx_free(PMEMoid oid);  
void *pmemobj_direct(PMEMoid oid);  
TX_BEGIN(PMEMobjpool *pop) / TX_END  
OID_NULL, OID_IS_NULL(PMEMoid oid)
```

libpmemobj

```
/* TX_STAGE_NONE */

TX_BEGIN(pop) {
    /* TX_STAGE_WORK */
} TX_ONCOMMIT {
    /* TX_STAGE_ONCOMMIT */
} TX_ONABORT {
    /* TX_STAGE_ONABORT */
} TX_FINALLY {
    /* TX_STAGE_FINALLY */
} TX_END

/* TX_STAGE_NONE */
```

libpmemobj c++

```
struct queue {  
    void  
    push(pmem::obj::pool_base &pop, int value){  
        pmem::obj::transaction::run(pop, [&] {  
            auto node = pmem::obj::make_persistent<queue_node>();  
            node->value = value;  
            node->next = nullptr;  
  
            if (head == nullptr) {  
                head = tail = node;  
            } else {  
                tail->next = node;  
                tail = node;  
            }  
        });  
    }  
};
```

libpmemblk

- Designed for array of blocks
 - updates to a single block are atomic

```
int main(int argc, char *argv[]){
    const char path[] = "/mnt/pmem_fsdax0/myfile";
    PMEMblkpool *pbp;
    size_t nelements;
    char buf[ELEMENT_SIZE];
    pbp = pmemblk_create(path, ELEMENT_SIZE, POOL_SIZE, 0666);
    if (pbp == NULL)
        pbp = pmemblk_open(path, ELEMENT_SIZE);
    }
    /* how many elements fit into the file? */
    nelements = pmemblk_nblock(pbp);
    strcpy(buf, "starting time");
    pmemblk_write(pbp, buf, 5)
    pmemblk_read(pbp, buf, 10);
    pmemblk_set_zero(pbp, 5);
    pmemblk_close(pbp);
}
```


libpmemlog

- B-APM resident log file
- Designed for append-mostly file
- Variable length entries

```
plp = pmemlog_create(path, POOL_SIZE, 0666);  
plp = pmemlog_open(path);  
data = "first thing";  
pmemlog_append(plp, data, strlen(data)) ;  
data = "second thing";  
pmemlog_append(plp, data, strlen(data));  
pmemlog_walk(plp, 0, printit, NULL);
```

pmempool

- The lib, blk, and obj functions use memory pools that can be created and managed using the mempool functions
- You can create a memory pool:

```
pmempool create obj --layout=simplekv -s 100M /mnt/pmem-fsdax0/simplekv-simple
```

- You can also interact with memory pools, i.e.:

```
pmempool info /mnt/pmem-fsdax0/simplekv-simple
```

Higher level functionality

- For object store and transactional support these libraries are the best choice
- Provide extra functionality that gives persistency support
 - Has associated performance overhead
- Similar to filesystem support for standard I/O
- If pure memory access is not the model, these are the functions to use
 - For pure memory access and manipulation the low level pmem will give best control and performance
- Files are a good starting point
 - But lack the granularity control and associated performance pmem gives

Persistent Memory Summary

- Optane hardware is complicated
- Performance is workload dependent (when isn't it?)
- Targeted usage will be required for the best performance
- I/O performance has been problematic for a while anyway
- In-node storage complicates data access and sharing

Object Stores

- Filesystems use Files as the container for blocks of data and the lowest level of metadata granularity
- Object stores use Objects as the container for data elements and the lowest level of metadata granularity
 - Allows individual pieces of data to be stored, indexed, and accessed separately
 - Allows independent read/write access to “blocks” of data

Object Stores

- Generally restricted interface
 - Put: Create a new object
 - Get: Retrieve the object
 - Removes the requirements for lots of functionality r.e. POSIX style I/O
- Traditionally objects are immutable
 - Once created cannot be changed
 - This removes the locking requirement seen for file writes
 - Makes updates similar to log-append filesystems, i.e. copy and update
- Object id generated when created
 - Used for access
 - Can be used for location purposes in some systems

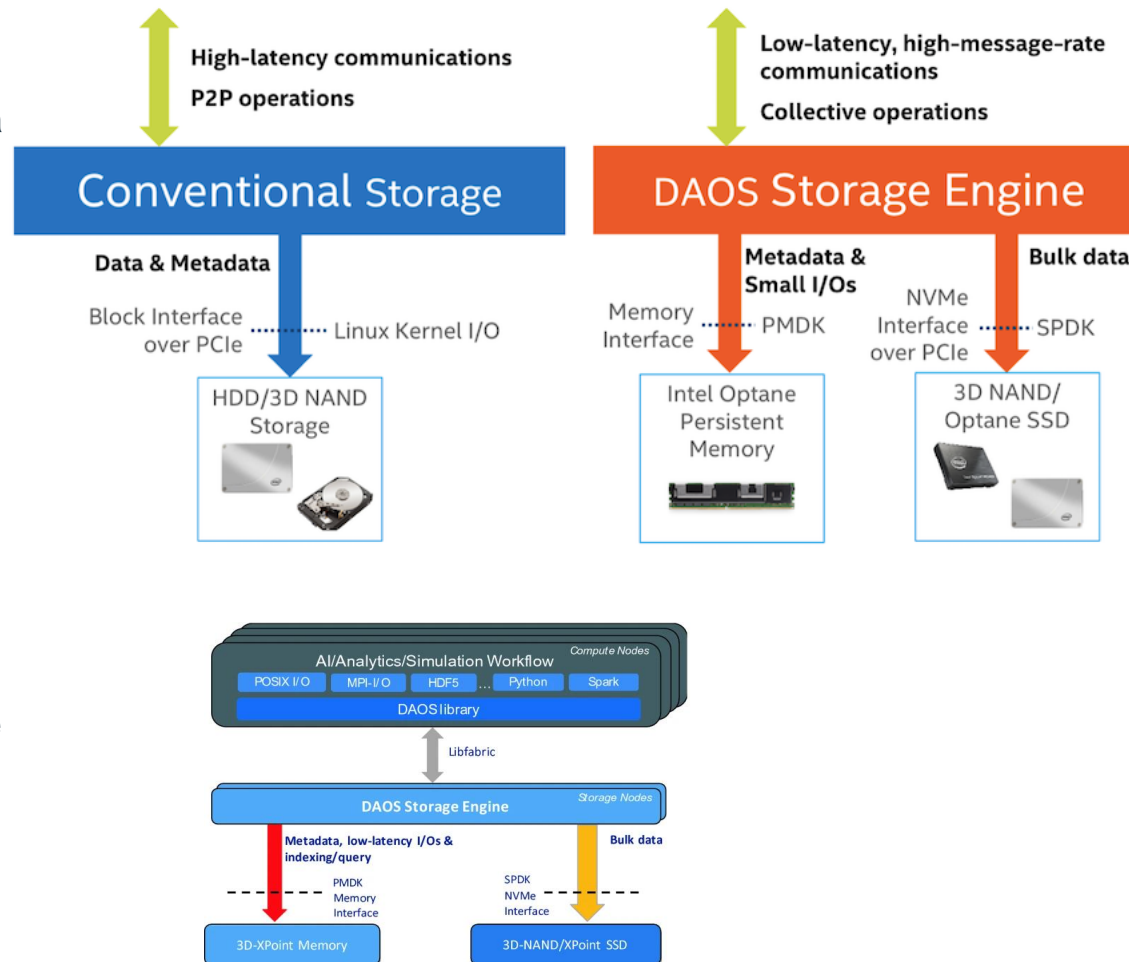
Object Stores

- Generally have helper services and interfaces
 - Manage metadata
 - Permissions
 - Querying
 - Etc...
- Distribution and redundancy etc... part of the complexity
 - Often eventual consistency
- Lots of complexity in implementations
- Often web interfaces as part of the Put/Get interface

DAOS

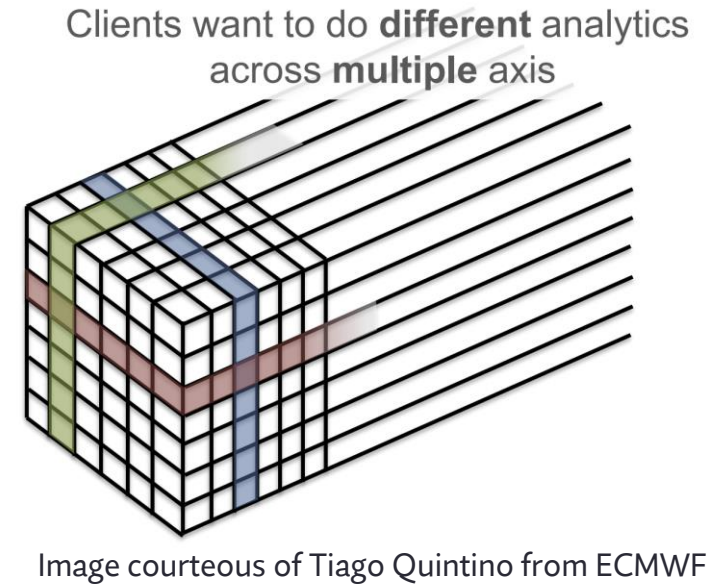
- Native object store on non-volatile memory and NVMe devices

- Pools
 - Define hardware range of data
- Containers
 - User space and data configuration definitions
- Objects
 - **Multi-level key-array** API is the native object interface with locality
 - **Key-value** API provides a simple key and variable-length value interface. It supports the traditional put, get, remove and list operations.
 - **Array API** implements a one-dimensional array of fixed-size elements addressed by a 64-bit offset. A DAOS array supports arbitrary extent read, write and punch operations.



Object store

- Software approach to granular data access
- Enabling generating and querying data on different “dimensions”
- Enabling data sizes to vary whilst maintaining performance
- Enable “legacy” interfaces



Practical Setup

- <https://github.com/NGIOproject/PMTutorial>
- Take IOR and STREAMS source code
- Run on the prototype
- Prototype is available at:

```
ssh ngio-adrianj.epcc.ed.ac.uk
```

- Then

```
ssh hydra-vpn.epcc.ed.ac.uk
```

- Then

```
ssh nextgenio-login2
```

- Using your guest account
- Practical source code is available at:

```
/home/nx01/shared/pmtutorial/exercises
```

- Using Slurm as the batch system

```
sbatch --reservation=nx01_59 scriptname.sh
```

Or

```
srun --reservation=nx01_59 -N 1 --nvram-options=1LM:1000 --pty /bin/bash
```

Emerging_interfaces

```
module load compiler  
module load mpi  
src/C/Makefile.config  
mpicc -> mpiicc
```