LOW LEVEL PERSISTENT MEMORY PROGRAMMING

Adrian Jackson

EPCC

a.jackson@epcc.ed.ac.uk

@adrianjhpc

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

libmemkind

- If persistence is **not** required can use libmemkind
  - Access large memory space
- Provides allocation on persistent memory
- Requires a directory mounted on persistent memory with DAX approach to work
  - File is transparent
  - Removed when application terminates
  - Used exactly as DRAM
  - Performance will vary depending on cache usage

libmemkind

```
#include <memkind.h>
struct memkind *my_data;
size_t array_size = 1024*1024*1024;

memkind_create_pmem("/mnt/fsdax0/", array_size, &my_data);

/* Create a float array of 1024 elements */
float *array_a = (float*)memkind_malloc(my_data, sizeof(float)*1024);

array_a[0] = 1.5;

memkind_free(my_data, array_a);
```

libmemkind

- New version while have automatic NUMA awareness
- No longer requires `memkind_create_pmem`
- Simply use (or something similar):

```
memkind_malloc(my_data, sizeof(float)*1024, MEMKIND_DAX_KMEM);
```

- Uses daxctl (ndctl functionality)

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();
}
```

pmem_is_pmem

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

- Returns true if the whole address range is on B-APM
- Can then use `pmem_persist` function rather than `msync/pmem_msync`
- Enables code to be written that works without B-APM hardware

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)

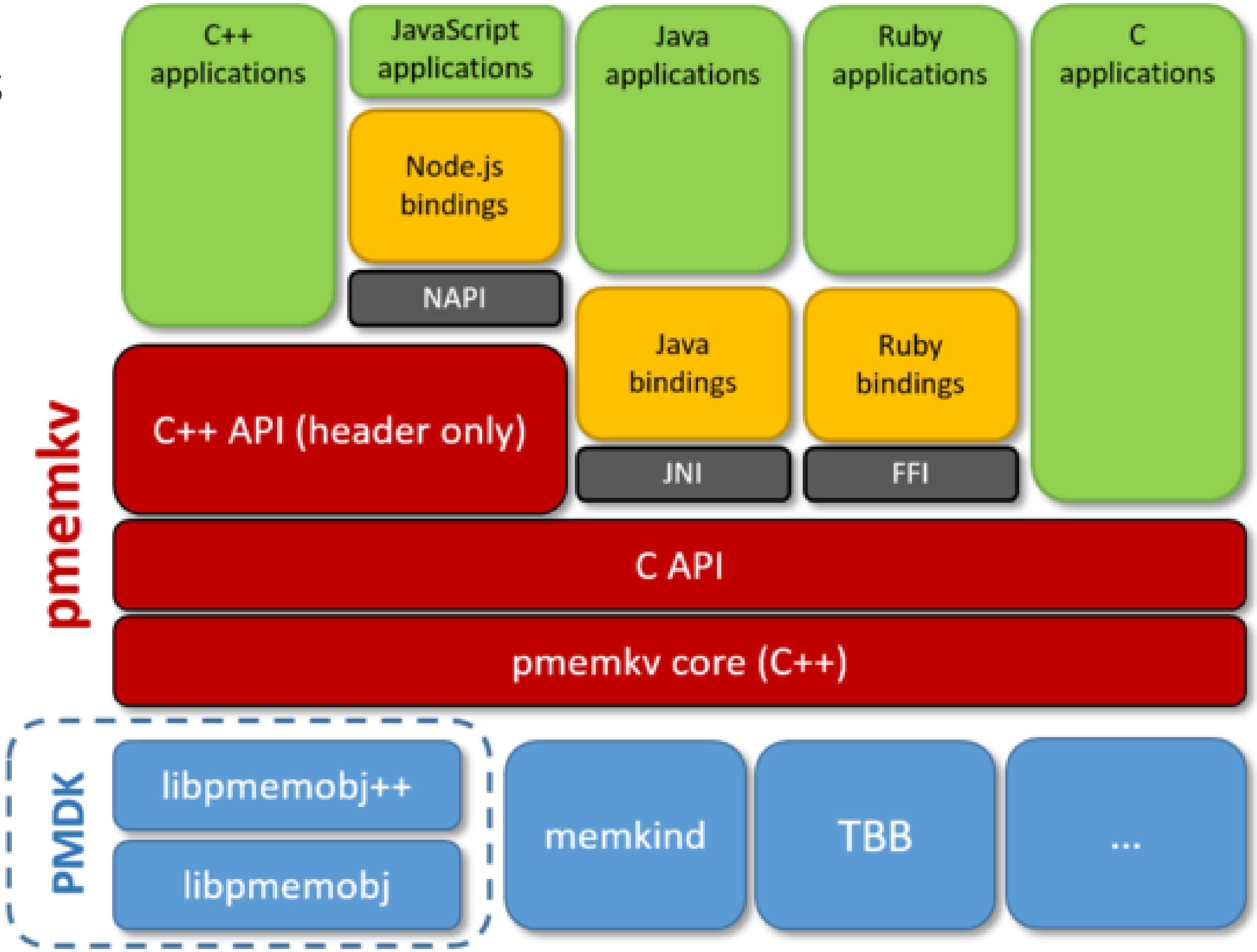
- Basic persistence functionality provided by the PMDK pmem library
  - Required if you want to use the persistent feature of persistent memory
- If large memory is all that is required libmemkind is available
- pmem defers responsibility to the programmer to ensure persistence is properly handled
  - 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("vsmap","{\"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
vsmap	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