



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, ipde, n), i=-1, imax1), j=-  
1, jmax1), k=-1, kmax1), ipde=1, npde)  
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 programming



- 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

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 mapped files



- 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 approach



- To utilise NVRAM memory functions are need to allocated, deallocate, and persist memory in the address ranges of the NVDIMMs
- mmap approach is followed
 - Except the NVRAM data is not mapped into DRAM
 - And there is no initial read of data because it is already in NVRAM
 - And persisting is done on cache lines
- Functions to help run both on NVRAM and non-NVRAM systems
- PMDK has various different approaches for using B-APM
 - <https://github.com/pmem/pmdk>

libmemkind



- If persistence is **not** required can use libmemkind
- Provides allocation on B-APM
- Requires a directory mounted on B-APM 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 B-APM 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/dax12.0
- 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

pmem_persist



```
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 operation functions



```
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;  
}
```

Practical



- Implement Streams with libmemkind functionality and pmem functionality
- Compare the performance
- Consider different options for arrays to place in B-APM and when to persist the data
- Write an application that persists data to pmem then another that reads that data back and checks it is correct