SC20

Everywhere we are | more than hpc.

Higher Level Persistent Memory Programming

Adrian Jackson
EPCC
a.jackson@epcc.ed.ac.uk
@adrianjhpc

epcc

Barcelona Supercomputing Center
Centro Nacional de Supercomputación

PMDK

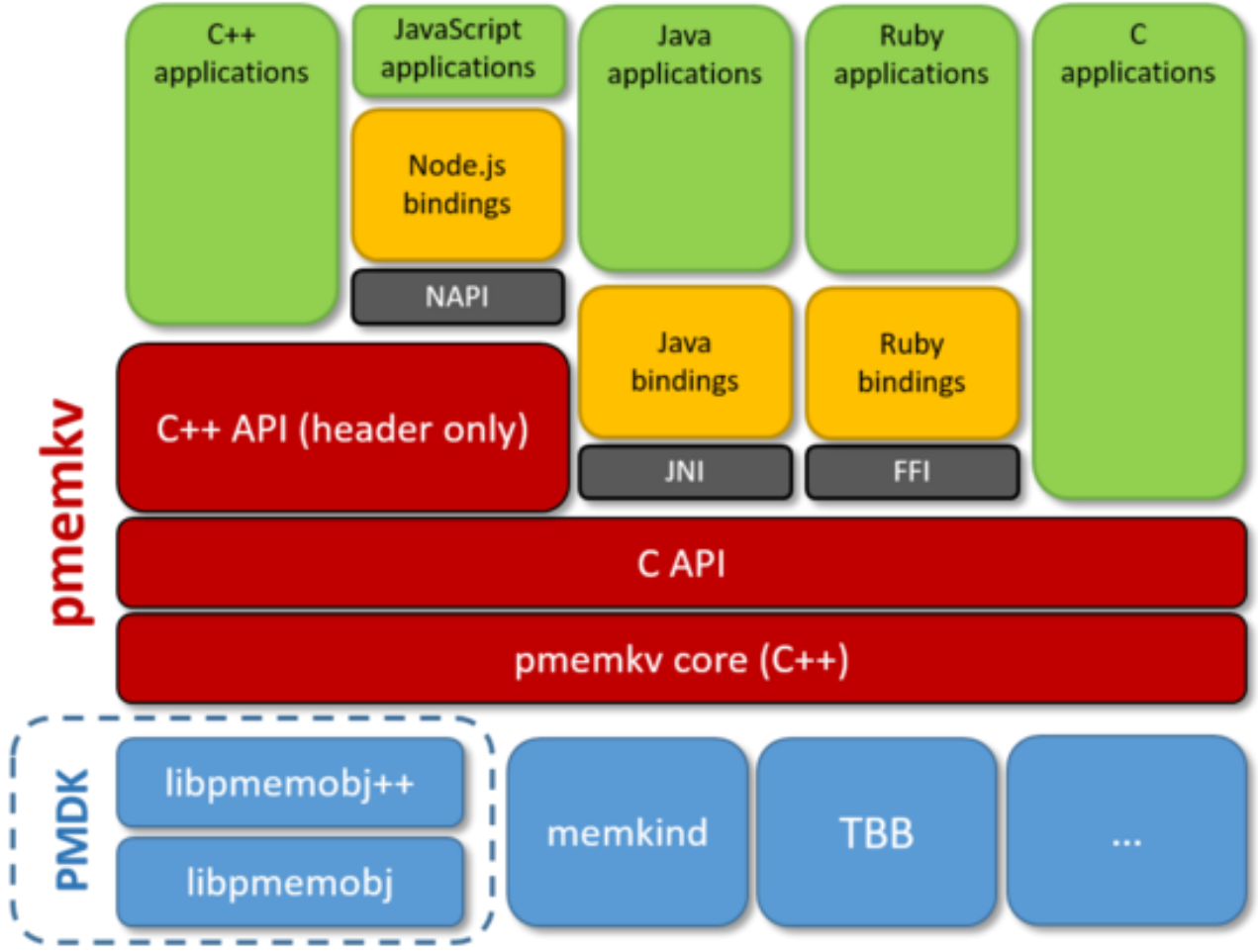
- Provides higher level programming functionality
- How people mainly expect you to be programming B-APM
- Transactional memory
- Key value stores
- Memory pools
- Manages the persistence nasties for you

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