PERSISTENT MEMORY PROGRAMMING MADE EASY WITH PMEMKV

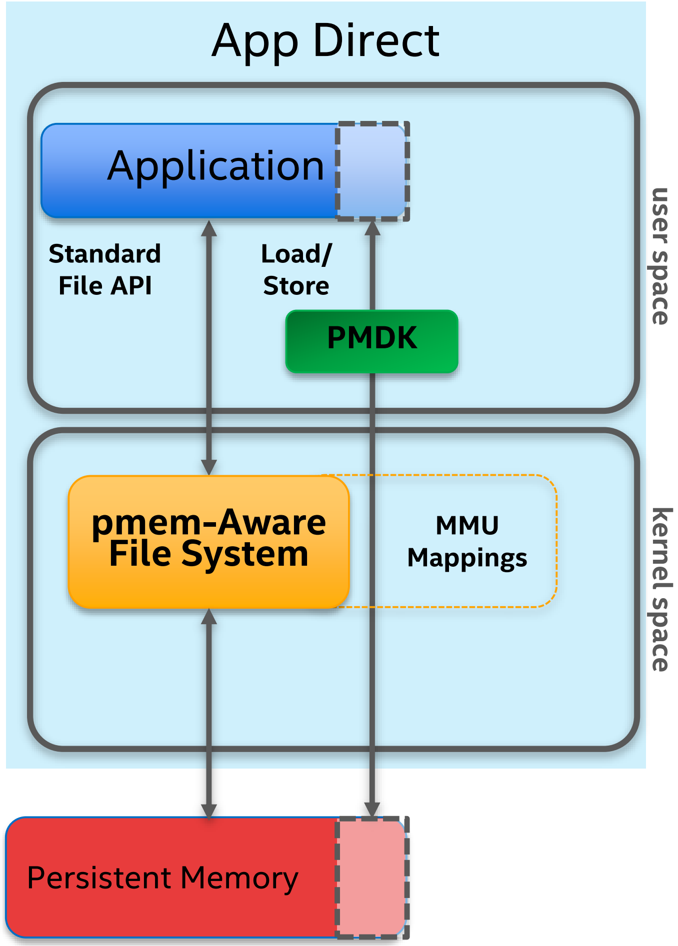
AGENDA

- Why pmemkv?
  - Persistent Memory programming is difficult
  - key-value store
- pmemkv design
  - goals for pmemkv
  - architecture
  - configuration
  - life-cycle (persistent libpmemobj-based engines)
- Engines
  - overview
  - multiple engines within the same memory pool
- Language bindings
- pmemkv is simple!
  - API
  - C++ example
  - NodeJS example
- Latencies and performance
- Q&A

WHY PMEMKV?

Why pmemkv?

Persistent Memory programming is difficult



- Different modes for using Persistent Memory:
  - Memory Mode
  - Storage over App Direct
  - App Direct
- A pmem-aware file system exposes persistent memory to applications as files
- In-place persistence (no paging, context switching, interrupts, nor kernel code executes)
- Byte addressable like memory (Load/store access, no page caching)
- Cache Coherent

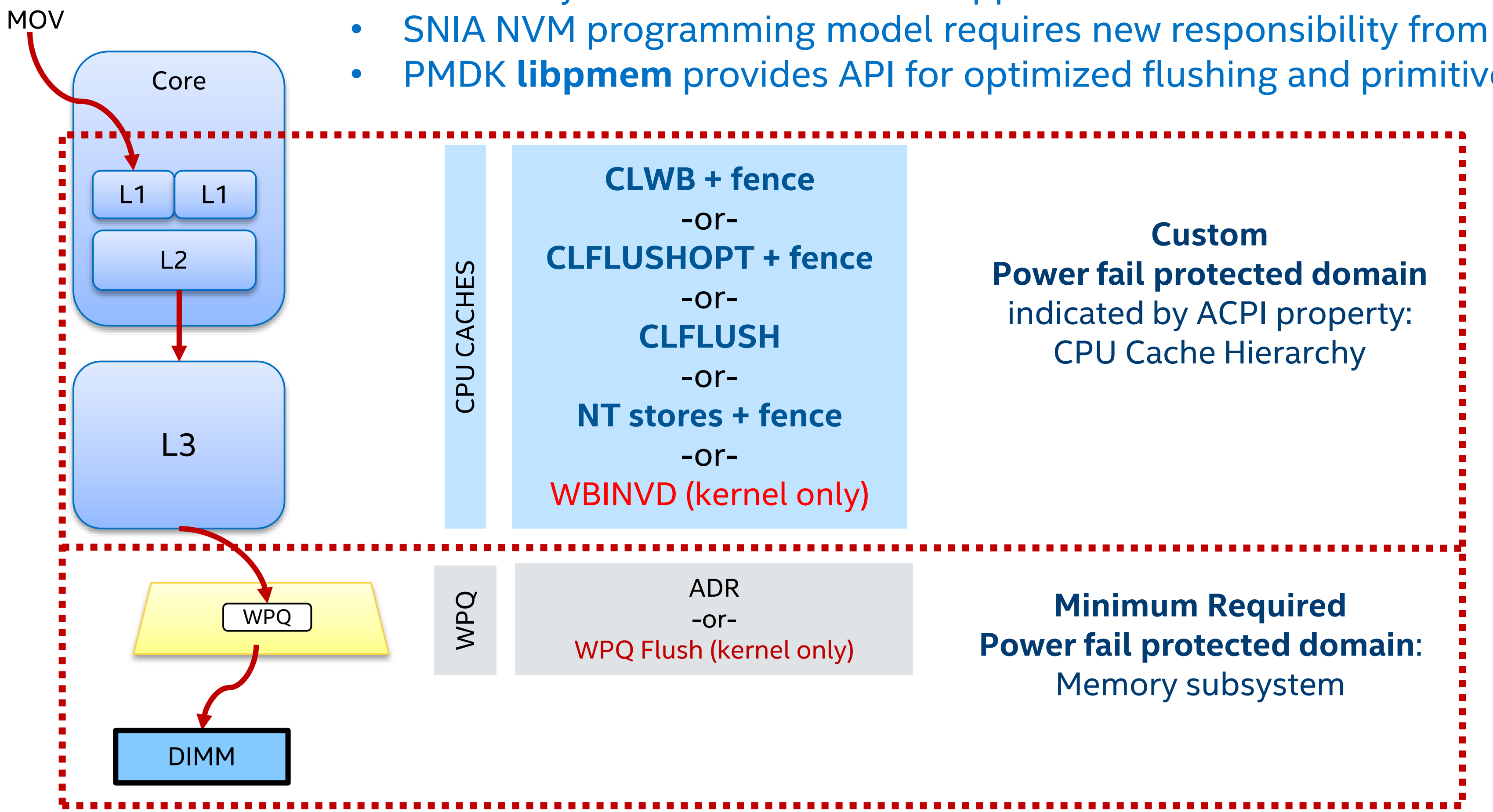
```
fd = open("/my/file", O_RDWR);
…
base = mmap(NULL, filesize,
                        PROT_READ|PROT_WRITE,
                        MAP_SHARED_VALIDATE|MAP_SYNC, fd, 0);
close(fd);
…
base[100] = 'X';
strcpy(base, "hello there");
msync(…);
…
```

It looks easy! But…

Why pmemkv?

Persistent Memory programming is difficult

- Where is your data when crash happens?
- SNIA NVM programming model requires new responsibility from user application: **flushing**
- PMDK **libpmem** provides API for optimized flushing and primitives like memcpy/memmove/memset



Why pmemkv?

Persistent Memory programming is difficult

```
open(…);
mmap(…);
strcpy(pmem, "Hello, World!");
pmem_persist(pmem, 14);
```

Crash

Result?

```
1. "\0\0\0\0\0\0\0\0\0\0..."
2. "Hello, W\0\0\0\0\0\0..."
3. "\0\0\0\0\0\0\0\0orld!\0"
4. "Hello, \0\0\0\0\0\0\0\0"
5. "Hello, World!\0"
```

- pmem_persist is faster than msync(), but it is still not transactional
- SNIA NVM programming model requires new responsibility from user application: **consistency**
- PMDK **libpmemobj** provides transactional API, Persistent Memory allocator etc.

```
struct data {
    p<long long> x;
}

auto pop = pool<data>::("/path/to/poolfile", "layout string");
auto datap = pop.root();

transaction::run(pop, [&]{
    datap->x = 5;
});
```

Learn more about libpmemobj and C++ programming during session: “Creating C++ Apps with libpmemobj”

Why pmemkv?

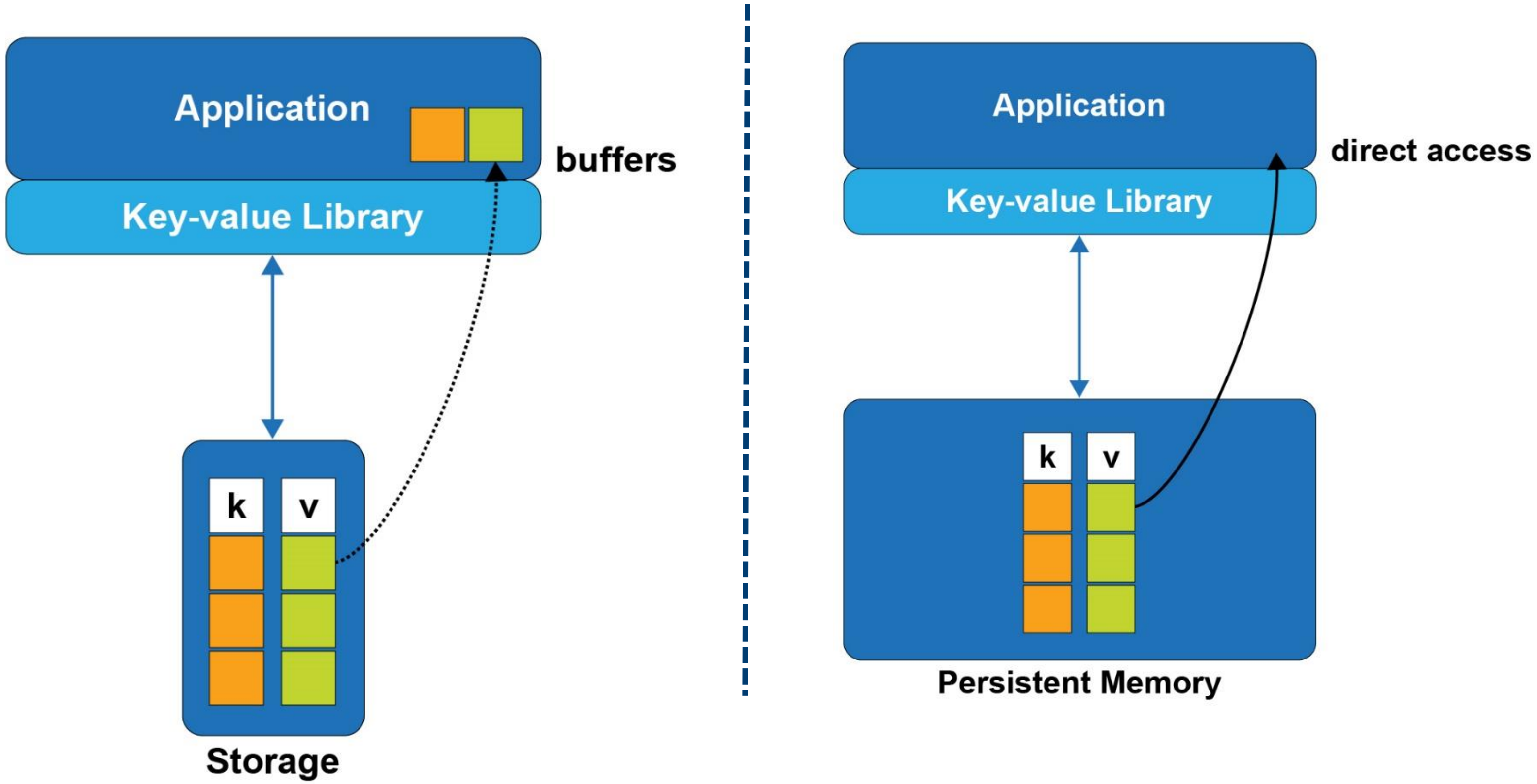
Q: How to make Persistent Memory programming easier?
A: Local Key-Value data store

- API flexibility increases complexity
  - API flexibility not always desired
- Usually the bigger barrier to adoption, the better performance gains
  - Don’t have to be true for some specific workloads

- Large addressable market of cloud developers for an easy KV store
  - Data stored in cloud will be the majority of all stored data in nearby future
- Key-Value data store provides straightforward API which can easily utilize Persistent Memory advantages
  - Nothing new to learn in order to start using Persistent Memory in efficient way
- Simple API makes creation of different language bindings relatively easy
  - Important in cloud native computing, where many high-level languages are being used

Why pmemkv?

- Key-value store can take advantage from persistence and big capacity of Persistent Memory
- Key-value store can utilize Persistent Memory byte addressability
  - huge performance gain for relatively small key and values



PMEMKV DESIGN

pmemkv design

goals for pmemkv

Technical:

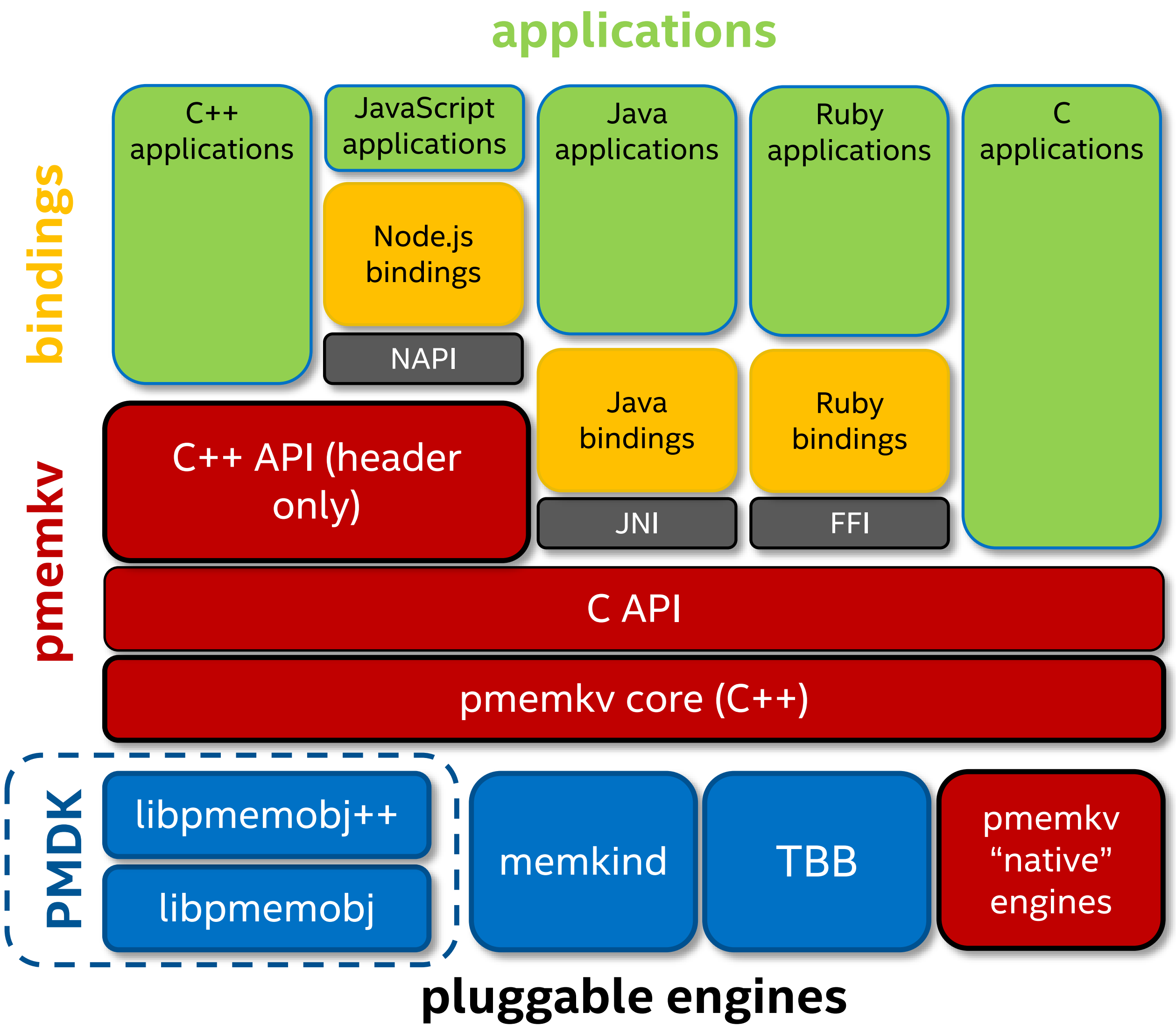
- Local key/value store (no networking)
- Idiomatic language bindings
- Simple, familiar, bulletproof API
- Easily extended with new engines
- Optimized for persistent memory (limit copying to/from DRAM)
- Flexible configuration, not limited to a single storage algorithm
- Generic tests

Community:

- Open source, developed in the open and friendly licensing
  - https://github.com/pmem/pmemkv
- Outside contributions are welcome
- Intel provides stewardship, validation on real hardware, and code reviews
- Standard/comparable benchmarks

pmemkv design
architecture

- pmemkv core is a frontend for engines
  - Core implementation written in C++, not related to Persistent Memory
- Pluggable engines
  - Some engines are implemented in pmemkv, some engines are imported from external projects
  - Persistent engines are implemented with libpmemobj (PMDK)
- Native API for pmemkv is written C/C++
- pmemkv design allows for easy integration with high-level language bindings



pmemkv design
configuration

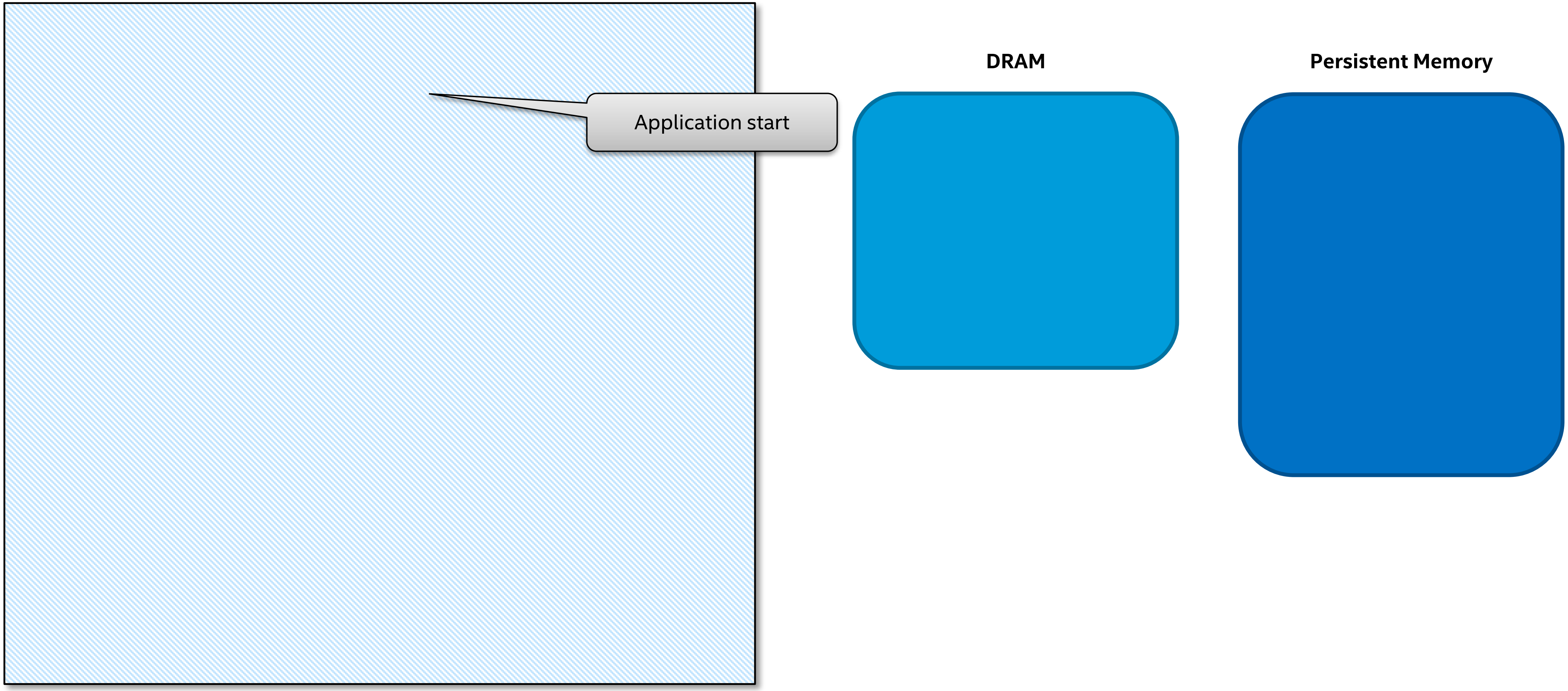
- Flexible configuration API
  - Works with different kinds of engines
- Every engine has documented supported config parameters individually
- Unordered map
  - Takes name configuration value as a k-v pair
- Supported configuration types:
  - int64/uint64/double
  - string
  - Arbitrary data (pointer and size)
- Resides on stack
  - Takes optional destructor as an additional parameter if custom configuration parameter allocates memory

Typical config structure example for libpmemobj-based engines

```
config cfg;

status s = cfg.put_string("path", path);
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);

s = cfg.put_uint64("force_create", 1);
assert(s == status::OK);
```

pmemkv design

life-cycle (persistent engines based on libpmemobj)

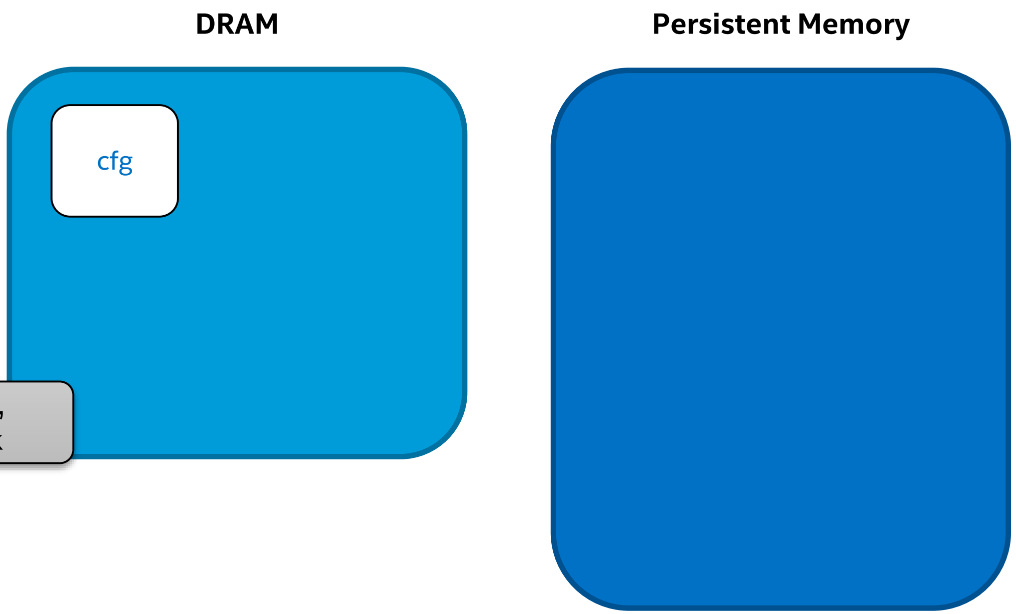


pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string("path", "/daxfs/file");
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);
```

config structure, resides on stack

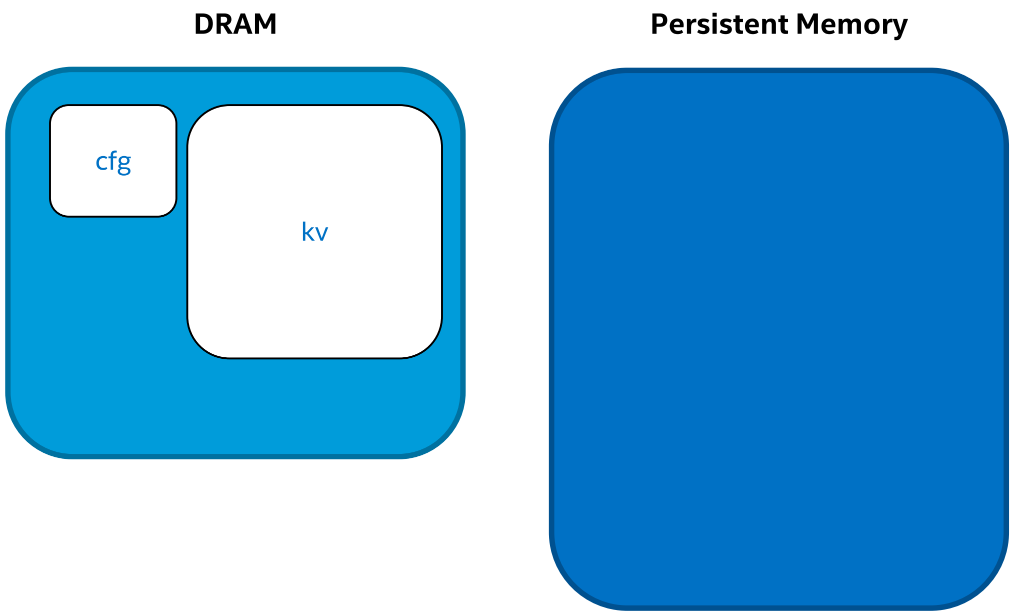


pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string(“path", “/daxfs/file”);
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);

db *kv = new db();
```

db object – volatile object for managing engine

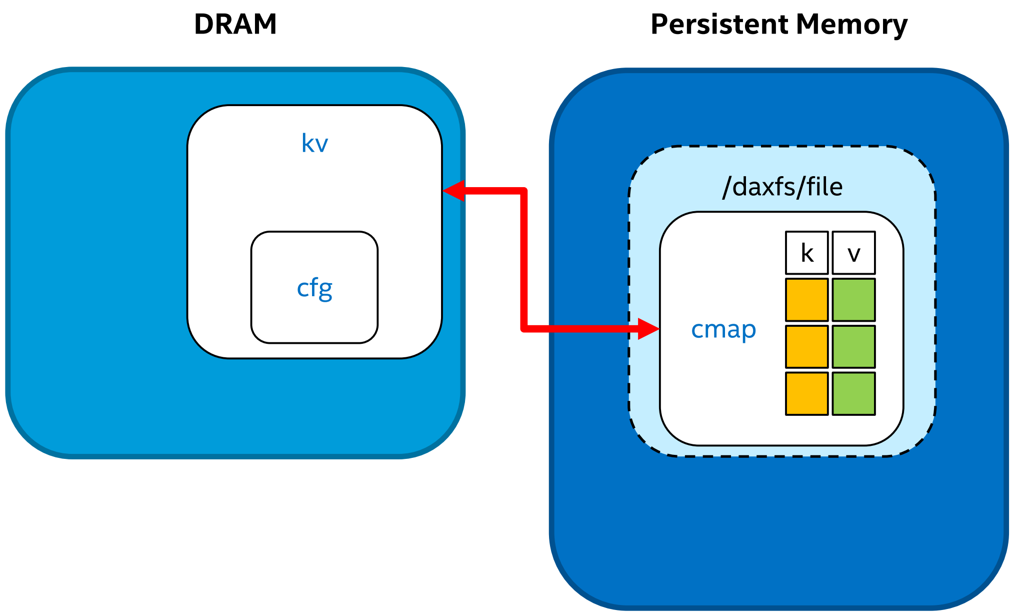


pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string("path", "/daxfs/file");
assert(s == status::OK);

s = cfg.put_uint64("size
assert(s == status::OK);

db *kv = new db();

if (kv->open("cmap", cfg) != status::OK) {
    std::cerr << db::errormsg() << std::endl;
    return 1;
}
```

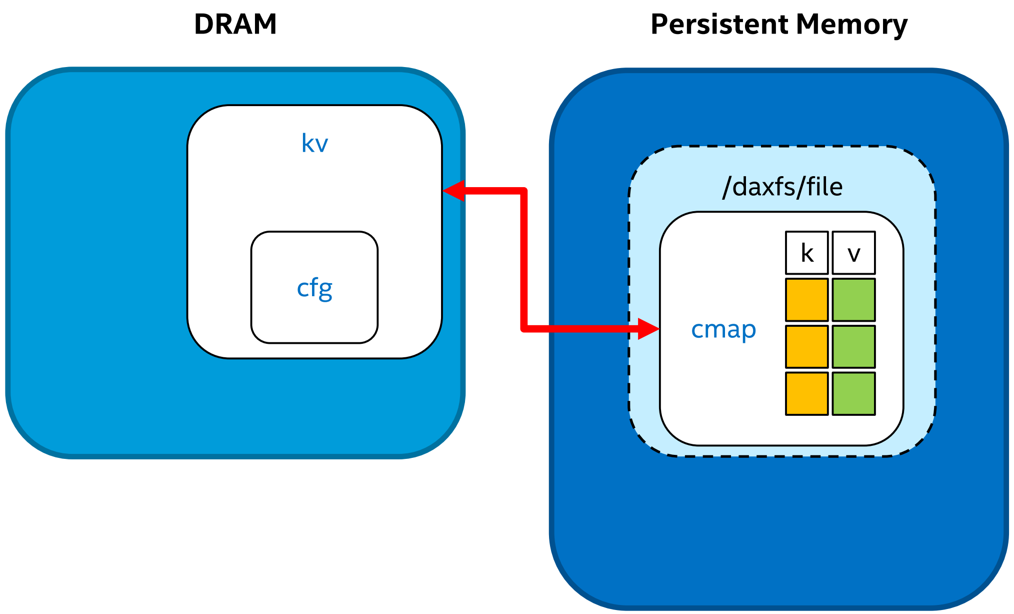
kv.open()
- creates/opens persistent memory pool
- checks consistency and perform recovery
- takes ownership for cfg structure



pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string("path", "/daxfs/file");
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);

db *kv = new db();

if (kv->open("cmap", cfg) != status::OK) {
    std::cerr << db::errormsg() << std::endl;
    return 1;
}

// busy work here
```

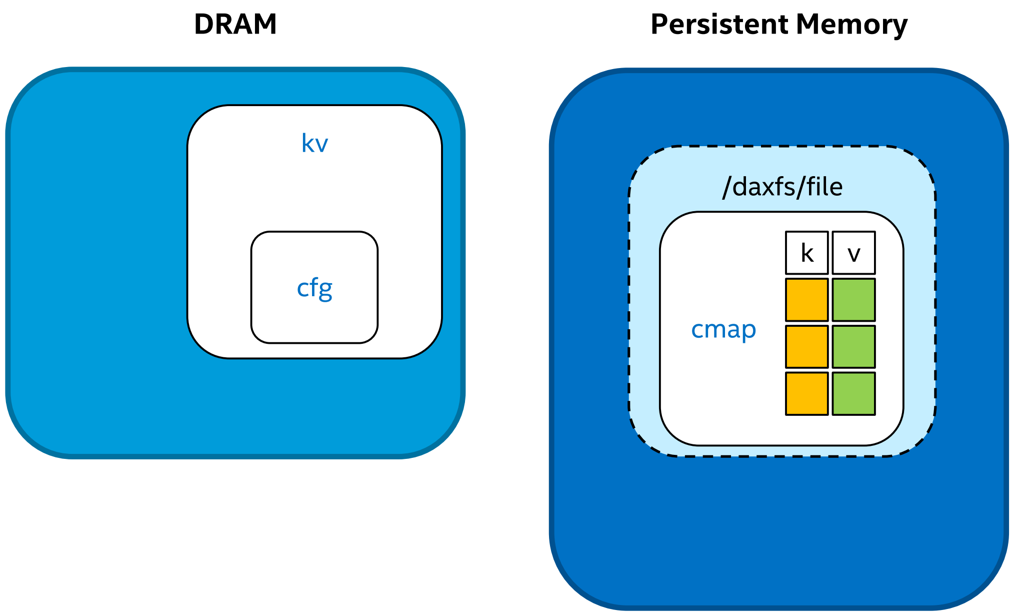


pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string("path", "/daxfs/file");
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);

db *kv = new db();

if (kv->open("cmap", cfg) != status::OK) {
    std::cerr << db::errormsg() << std::endl;
    return 1;
}

// busy work here

kv->close();
```

kv.close()
- close database connection
- Persistent Memory data remain saved

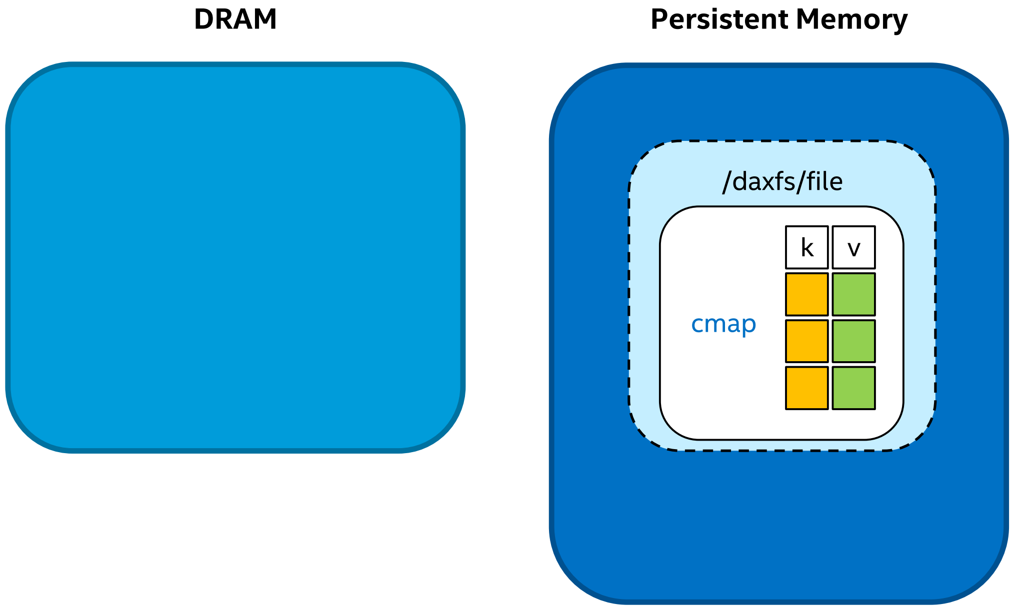


pmemkv design

life-cycle (persistent engines based on libpmemobj)

```
config cfg;

status s = cfg.put_string("path", "/daxfs/file");
assert(s == status::OK);

s = cfg.put_uint64("size", SIZE);
assert(s == status::OK);

db *kv = new db();

if (kv->open("cmap", cfg) != status::OK) {
    std::cerr << db::errormsg() << std::endl;
    return 1;
}

// busy work here

kv->close();

delete kv;
```

Safety deletion of volatile data



ENGINES

Engines

overview

- Engine contributions are welcome!
- Types:
  - ordered/unordered
  - persistent/volatile
  - concurrent/single threaded
- Engines are optimized for different workloads & capabilities
- All engines work with all language bindings
- Generic tests for engines incl:
  - memcheck
  - helgrind/drd
  - pmemcheck
  - pmemreorder

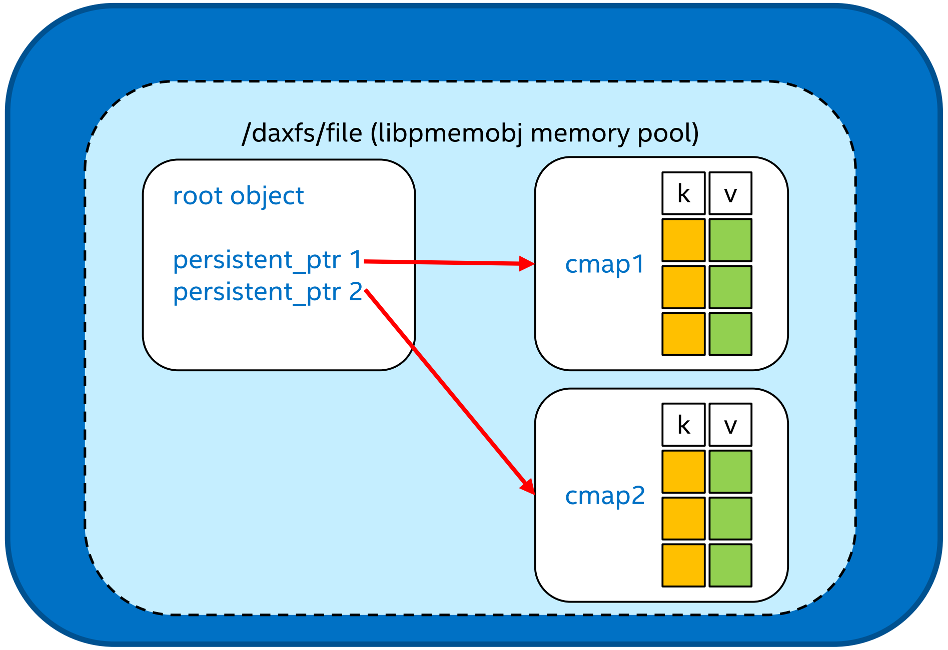
Engine Name	Description	Experimental?	Persistent?	Concurrent?	Sorted?
blackhole	Accepts everything, returns nothing	No	-	-	-
cmap	Concurrent hash map	No	Yes	Yes	No
vsmap	Volatile sorted hash map	No	No	No	Yes
vcmap	Volatile concurrent hash map	No	No	Yes	No
tree3	Persistent B+ tree	Yes	Yes	No	No
stree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

Engines

multiple engines within the same memory pool

- pmemkv API (config API) does not limit user for correlating single engine with single memory pool (libpmemobj)
- It is possible to pass persistent_ptr argument to config structure and attach engine to the pointer (ongoning work on generic API for libpmemobj-based engines)

Persistent Memory



Learn more about libpmemobj and persistent_ptr during session:
"Creating C++ Apps with libpmemobj"

- Engines are reachable from root object

Engines

multiple engines within the same memory pool

```
struct Root {
        pmem::obj::persistent_ptr<PMEMoid> ptr1;
        pmem::obj::persistent_ptr<PMEMoid> ptr2;
};
// libpmemobj setup here
config cfg_1;
config cfg_2;
status ret = cfg_1.put_object("oid", &(pop.root()->ptr1),
nullptr);
assert(ret == status::OK);
ret = cfg_2.put_object("oid", &(pop.root()->ptr2), nullptr);
assert(ret == status::OK);

db *kv_1 = new db();
status s = kv_1->open("cmap", std::move(cfg_1));
assert(s == status::OK);

db *kv_2 = new db();
s = kv_2->open("cmap", std::move(cfg_2));
assert(s == status::OK);
```

Learn more about libpmemobj during session:
“Creating C++ Apps with libpmemobj”

Prototyped API for using pmemkv with libpmemobj++ simultaneously (implementation work ongoing)

LANGUAGE BINDINGS

Language bindings

Simple API = easy to implement high-level language bindings with small performance overhead

- Currently 4 available language bindings for pmemkv:
  - Java https://github.com/pmem/pmemkv-java
  - NodeJS https://github.com/pmem/pmemkv-nodejs
  - Ruby https://github.com/pmem/pmemkv-ruby
  - Python https://github.com/pmem/pmemkv-python
- Their APIs are not functionally equal to native C/C++ counterpart
  - Configuration possible only by JSON string passed to open() function
  - Multiple engines within single memory pool not possible
  - Above API gaps are under development

PMEMKV IS SIMPLE!

pmemkv is simple!
API

- Well understood key-value API
  - Nothing new to learn
  - Inspired by rocksDB and levelDB

- Life-cycle API
  - open()/close()

- Operations API
  - put(key, value)
  - get(key, value/v_callback)
  - remove(key)
  - exists(key)

- other
  - errormsg()

- Iteration API
  - count_all()
  - get_all(kv_callback)
- +range versions of above for ordered engines
  - below/above/between

pmemkv is not limited to the API above – in future, specific engines might provide extensions and methods like batch_update()

pmemkv is simple!

C++ example

```
config cfg;
// setup config here
status ret = kv.open("cmap", cfg);
assert(ret == status::OK);

ret = kv.put("John", "123-456-789");
assert(ret == status::OK);

std::string number;
ret = kv.get("John", &number);
assert(ret == status::OK);

ret = kv.get_all([](string_view name, string_view num) {
         std::cout << name.data() << " " << num.data() << std::endl;
});
assert(ret == status::OK);
assert(kv.exists("John") == status::OK);

ret = (kv.remove("John");
assert(ret == status::OK);

kv.close();
```

Get value by copying to DRAM

Direct access to Persistent Memory by callback; it make sense because we need to lock others from removing the value while someone has a direct pointer to it.

pmemkv is simple!

NodeJS example

```
const db = new Database('cmap', '{"path":"/daxfs/kvfile","size":1073741824}');

db.put('John', '123-456-789');

assert(db.get('John') === '123-456-789');

db.get_all((k, v) => console.log(`name: ${k}, number: ${v}`));

db.remove('John');

assert(!db.exists('John'));

db.stop();
```

- Similar simplicity with other high-level language bindings

pmemkv is simple!
NodeJS example

```
const db = new Database('cmap', '{"path":"/daxfs/kvfile","size":1073741824}');

db.put('John', '123-456-789');

assert(db.get('John') === '123-456-789');

db.get_all((k, v) => console.log(`name: ${k}, number: ${v}`));

db.remove('John');

assert(!db.exists('John'));

db.stop();
```

- Similar simplicity with other high-level language bindings

pmemkv is simple!

Java example

```
import io.pmem.pmemkv.Database;

public class BasicExample {
    public static void main(String[] args) {
        Database db = new Database("vsmap", "{\"path\":\"/dev/shm\",
\"size\":1073741824}");

        db.put("John", "123-456-789");
        assert db.get("John").equals("123-456-789");

                db.get_all((String k, String v) -> System.out.println("name: " + k + ",
number: " + v));

        db.remove("John");
        assert !db.exists("John");

        db.stop();
    }
}
```

pmemkv is simple!

Python example

```
from pmemkv import Database

print ("Starting engine")
db = Database(r"vsmap", '{"path":"/dev/shm","size":1073741824}')

db.put(r"key1", r"value1")
assert db.count_all() == 1
assert db.get(r"key1") == r"value1"

db.put(r"key3", r"value3")
db.get_keys_strings(lambda k: print ("  visited: {}".format(k.decode())))

db.remove(r"key1")
assert not db.exists(r"key1")

db.stop()
```

pmemkv is simple!

Ruby example

```ruby
require '../lib/pmemkv/database'

db = Database.new('vsmap', "{\"path\":\"/dev/shm\",\"size\":1073741824}")

db.put('key1', 'value1')
assert db.count_all == 1

assert db.get('key1').eql?('value1')

db.put('key2', 'value2')
db.put('key3', 'value3')
db.get_keys {|k| puts "  visited: #{k}"}

db.remove('key1')
assert !db.exists('key1')

db.stop
```

LATENCIES AND PERFORMANCE

Latencies and performance

- Language bindings
  - number of round trips between high-level language & native code
  - Create high-level object (string, byte[], reference type, callback/closure)
  - Translate bytes to UTF-8
  - String interning, reference counting or GC
- pmemkv core (native code)
  - Searching indexes in DRAM
  - Updating indexes in DRAM
  - Managing transactions
  - Allocating persistent memory
- Persistent Memory
  - HW read and write latency
- Performance varies based on traffic pattern
  - Contiguous 4 cacheline (256B) granularity vs. single random cacheline (64B) granularity
  - Read vs. writes

Latencies and performance

cmap performance

- pmemkv_tools is a separate github repository with benchmark tool inspired by db_bnch
  - https://github.com/pmem/pmemkv-tools

- Test results for cmap (persistent concurrent hashmap)
  - Throughput scales with a number of threads
  - P99 latency – flat

Q&A

experience
what's inside™