State of pmemkv

Paweł Karczewski
Software Engineer
Intel

Igor Chorążewicz
Software Engineer
Intel

Agenda

01 **Introduction to pmemkv**
Goals, architecture, engines

02 **Pmemkv is simple!**
API overview, C/C++ examples

03 **Language bindings overview**
Java, Python, Ruby, NodeJS

04 **Technical Overview**
Engines overview, lessons learned, ensuring data consistency

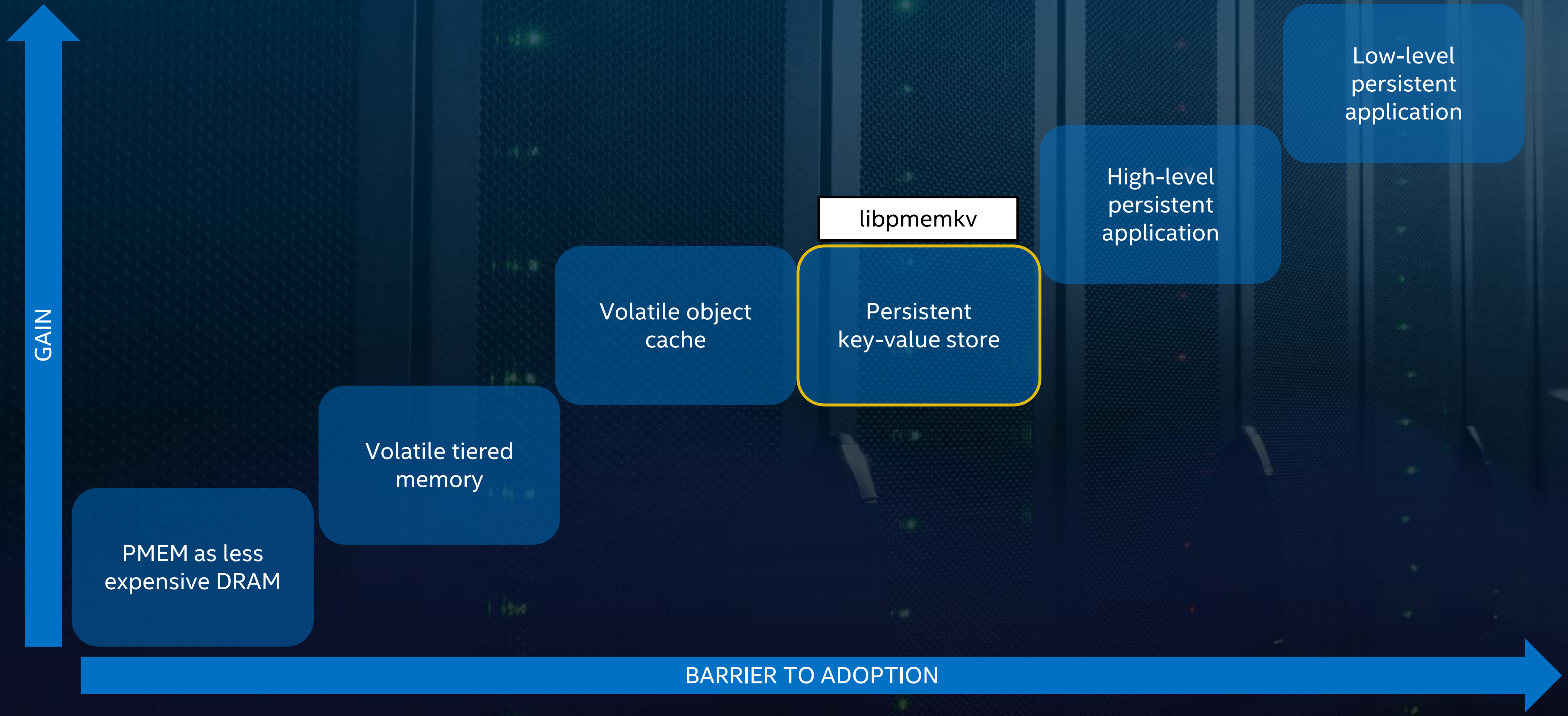
05 **Future plans**
Our next steps

Storage Performance Development Kit (SPDK)
Persistent Memory Development Kit (PMDK)
Intel® VTune™ Profiler

Virtual Forum

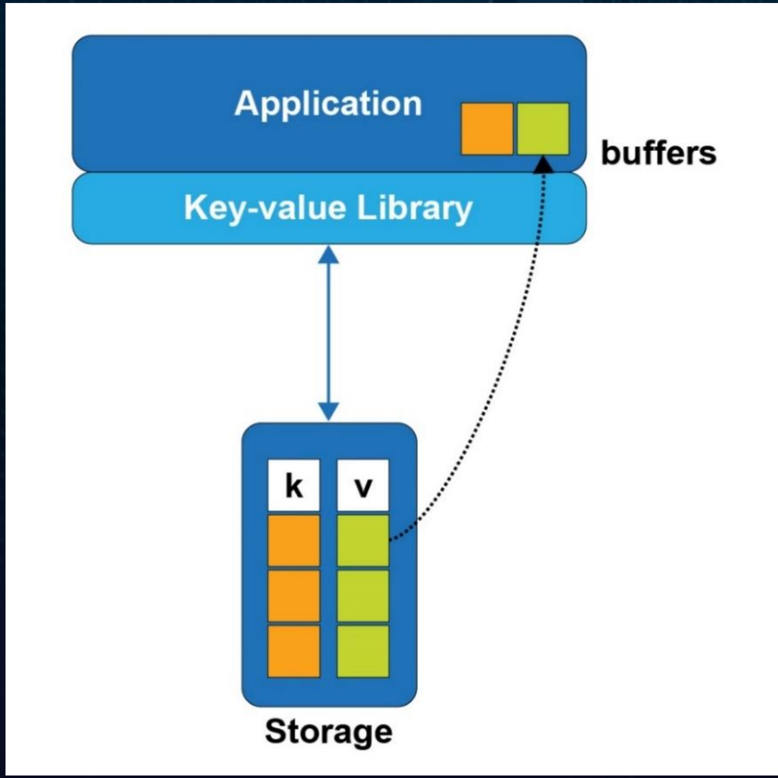
Introduction to pmemkv

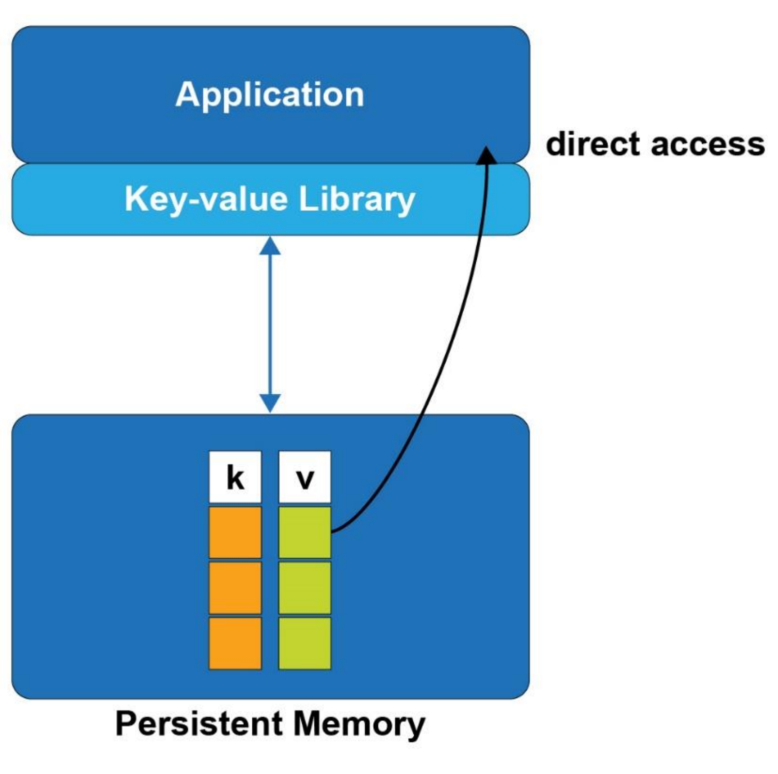
WHY PMEMKV?


GAIN
PMEM as less expensive DRAM
Volatile tiered memory
Volatile object cache
libpmemkv
Persistent key-value store
High-level persistent application
Low-level persistent application
BARRIER TO ADOPTION

WHY PMEMKV?

- Simple API allows to use persistent memory from high level languages
- 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 keys and values





DESIGN
Goals

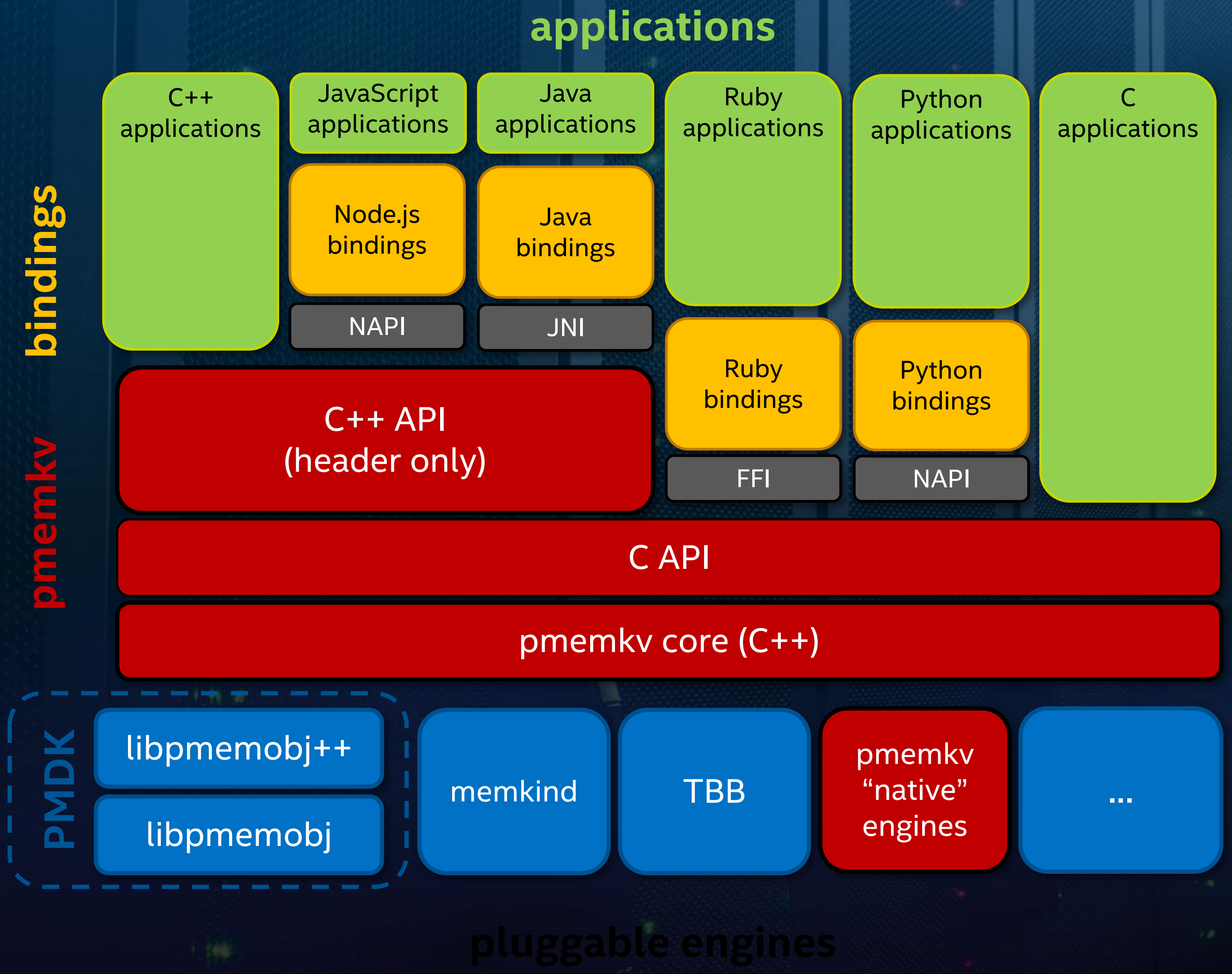
Technical:

- Local key/value store (no networking)
- Simple, familiar, bulletproof API
- Idiomatic language bindings
- Easily extended with new engines
- Flexible configuration, not limited to a single storage algorithm
- Generic tests
- Works in user space

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
- It's vendor-agnostic

DESIGN
Architecture



ENGINES
Overview

Engine Name	Description	Experimental	Concurrent	Sorted	Persistent
blackhole	Accepts everything, returns nothing	No (testing)	Yes	No	No
cmap	Concurrent hash map	No	Yes	No	Yes
vsmap	Volatile sorted hash map	No	No	Yes	No
vcmap	Volatile concurrent hash map	No	Yes	No	No
csmap	Concurrent sorted map	Yes	Yes	Yes	Yes
radix	Radix tree	Yes	No	Yes	Yes
tree3	Persistent B+ tree	Yes	No	No	Yes
stree	Sorted persistent B+ tree	Yes	No	Yes	Yes
robinhood	Persistent hash map with Robin Hood hashing	Yes	Yes	No	Yes
dram_vcmap	Volatile concurrent hash map placed entirely on DRAM	Yes (testing)	Yes	No	No

- Experimental engines are not yet a production quality (for various reasons) and their behavior may change. They are not included in build by default
- Test engines are delivered for benchmarking and testing reasons

Pmemkv is simple!

PMEMKV IS SIMPLE!

API

Full API documentation (C / C++): https://pmem.io/pmemkv

- 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
- Iterators
- Transactions

PMEMKV IS SIMPLE!

C++ example – Database configuration

```
pmem::kv::config cfg;

pmem::kv::status s = cfg.put_path("/mnt/pmem0/MyDatabase.pool");
s = cfg.put_size(SIZE);
assert(s == status::OK);
s = cfg.put_create_if_missing(true);
assert(s == status::OK);

/* Opening pmemkv database with 'cmap' engine */
db kv = pmem::kv::db();
s = kv.open("cmap", std::move(cfg));
ASSERT(s == status::OK);
```

PMEMKV IS SIMPLE!

C++ example – Direct data access via lambdas

```
s = kv.put("key1", "value1");
assert(s == status::OK);
s = kv.put("key2", "value2");
assert(s == status::OK);

/* Access value directly on PMEM */
s = kv.get([] (string_view v) {
    std::cout << v.data() << std::endl;
}
assert(s == status::OK);

/* Iterate over all key-value pairs directly on PMEM */
kv.get_all([](string_view k, string_view v) {
    std::cout << k.data() << " : " << c.data()<< std::endl;
    return 0;
});
s = kv.remove("key1");
assert(s == status::OK);
s = kv.exists("key1");
assert(s == status::NOT_FOUND);
```

PMEMKV IS SIMPLE!

C++ example – Direct data access via Iterators

- Experimental API
- read and write iterators
- Cannot hold simultaneously in the same thread more than one iterator.
- Holds lock per element

```
auto res_w_it = kv->new_write_iterator();
assert(res_w_it.is_ok());
auto &w_it = res_w_it.get_value();
/* seek to the element lower than "5" */
status s = w_it.seek_lower("5");
assert(s == status::OK);
do {
    std::string value_before_write =
    w_it.read_range().get_value().data();

    auto res = w_it.write_range();
    assert(res.is_ok());
    for (auto &c : res.get_value()) {
        c = 'x';
    }
    w_it.commit();
} while (w_it.next() == status::OK);
```

PMEMKV IS NOT SO SIMPLE!

C++ example – Transactions API, and why we need it?

- Allows grouping put/get/remove into single atomic action
- Provides ACID properties (no isolation for single threaded engines)
- Experimental API

```
auto result_tx = kv.tx_begin();
assert(result_tx.is_ok());

/* This function is guaranteed to not throw if is_ok is true */
auto &tx = result_tx.get_value();
s = tx.remove("key1");
s = tx.put("key2", "value2");

/* Until transaction is committed, changes are not visible */
assert(kv.exists("key1") == status::OK);
assert(kv.exists("key2") == status::NOT_FOUND);

s = tx.commit();
assert(s == status::OK);

assert(kv.exists("key1") == status::NOT_FOUND);
assert(kv.exists("key2") == status::OK);
```

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 (v. 1.0.1) https://github.com/pmem/pmemkv-java
  - Python (v. 1.0) https://github.com/pmem/pmemkv-python
  - NodeJS (v. 1.0) https://github.com/pmem/pmemkv-nodejs
  - Ruby (v. 0.9) https://github.com/pmem/pmemkv-ruby
- Their APIs are designed to fit into languages common practices

LANGUAGE BINDINGS
Java example

```java
Database<String, String> db = new Database.Builder<String, String>(ENGINE)
                .setSize(1073741824)
                .setPath("/dev/shm")
                .setKeyConverter(new StringConverter())
                .setValueConverter(new StringConverter())
                .build();

db.put("key1", "value1");
assert db.countAll() == 1;

assert db.getCopy("key1").equals("value1");

// Iterating existing keys
db.getKeys((k) -> System.out.println("  visited: " + k));
db.stop();
```

LANGUAGE BINDINGS
Java example

```java
class StringConverter implements Converter<String> {
    public ByteBuffer toByteBuffer(String entry) {
        return ByteBuffer.wrap(entry.getBytes());
    }

    public String fromByteBuffer(ByteBuffer entry) {
        byte[] bytes;
        bytes = new byte[entry.capacity()];
        entry.get(bytes);
        return new String(bytes);
    }
}
```

LANGUAGE BINDINGS
Python example

```python
import pmemkv

# Configuration dictionary
config = { "path":"/dev/shm",
    "size":1073741824 }

db = pmemkv.Database("vsmap", config)
db.put("key1", "value1")

# Get single value and key in lambda expression
key = "key1"
db.get(
    key,
    lambda v, k=key: print(
        f"key: {k} with value: " f"{memoryview(v).tobytes().decode()}"
    ),
)
db.stop();
```

LANGUAGE BINDINGS
NodeJS example

```
let config = {"path":"/dev/shm", "size":1073741824};
const db = new Database('vsmap', config, 'String');

try{
   db.put('key1', 'value1');
   db.put('key2', Buffer.from('value2'));
   assert(db.count_all === 1);
}
catch(e){
   if (e.status == constants.status.OUT_OF_MEMORY)
       console.log(e.message);
}

assert(db.get('key1') === 'value1’);
db.get_as_buffer('key2', (v) => {
   assert(v.toString() === 'value2’);
});

db.remove('key1’);
assert(!db.exists('key1'));
db.stop();
```

LANGUAGE BINDINGS
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
```

PERFORMANCE MEASUREMENTS

- pmemkv_bench is a separate GitHub repository with benchmark tool inspired by db_bench

  https://github.com/pmem/pmemkv-bench

Storage Performance Development Kit (SPDK)
Persistent Memory Development Kit (PMDK)
Intel® VTune™ Profiler

Virtual Forum

Technical overview

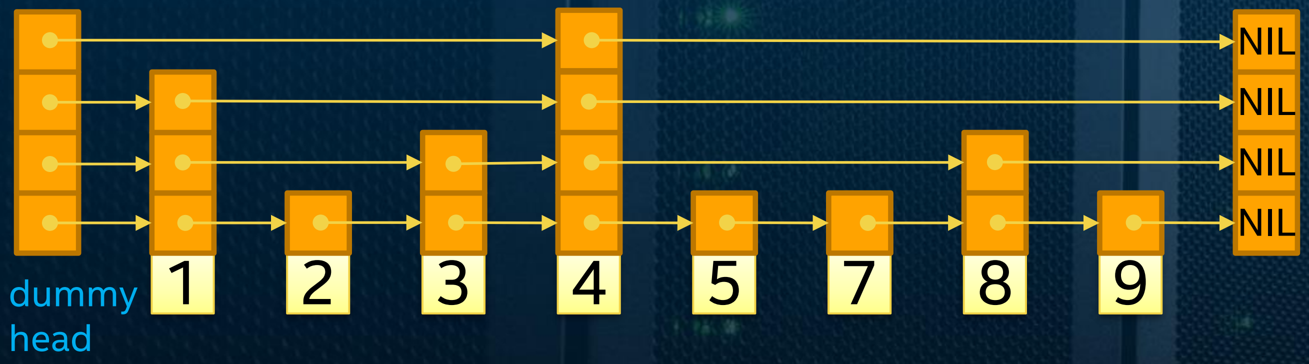
LIBPMEMOBJ-CPP

Transactional object store for Persistent Memory. It provides:

- ACID transactions
- Failure-atomic allocator
- General facilities useful for Persistent Memory programming
- Data structures optimized fo Persistent Memory
  - Vector, String, Array, Segment vector
  - Concurrent_map, Concurrent_hash_map
  - Radix tree

https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/OPODIS2006-BA.pdf

CSMAP: CONCURRENT SKIP LIST



Algorithm	Average	Worst
Space	O(n)	O(n log n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

- Multilayer linked list-like data structure.
    - The bottom layer is an ordinary ordered linked list.
    - Each higher layer acts as an "express lane" for the lists below.
- An element in layer i appears in layer i+1 with fixed probability p (in our case p = 1/2).
- Search is wait-free.
- Insert employs optimistic lock-based synchronization.

CSMAP: DELETE OPERATION

Our implementation does not support concurrent delete operation

- There is a way to logically delete a node from the skip list. But…
- There is a memory reclamation problem
  - We need to guarantee object life-time, while other threads accessing it
  - It is hard to solve without garbage collector
- There are possible solutions, but they might hurt Search/Insert performance
  - Hazard pointers
  - Epoch-based reclamation

DATA CONSISTENCY IN CONCURRENT LIST-LIKE DATA STRUCTURES

```
{
    manual tx;

    // allocate new node
    auto ptr = make_persistent(...);

    // insert node to the list
    atomic_store(list>next, ptr);

// other threads will see uncomitted state

    commit();
}
```

- Cannot easily use atomic instruction within a transaction

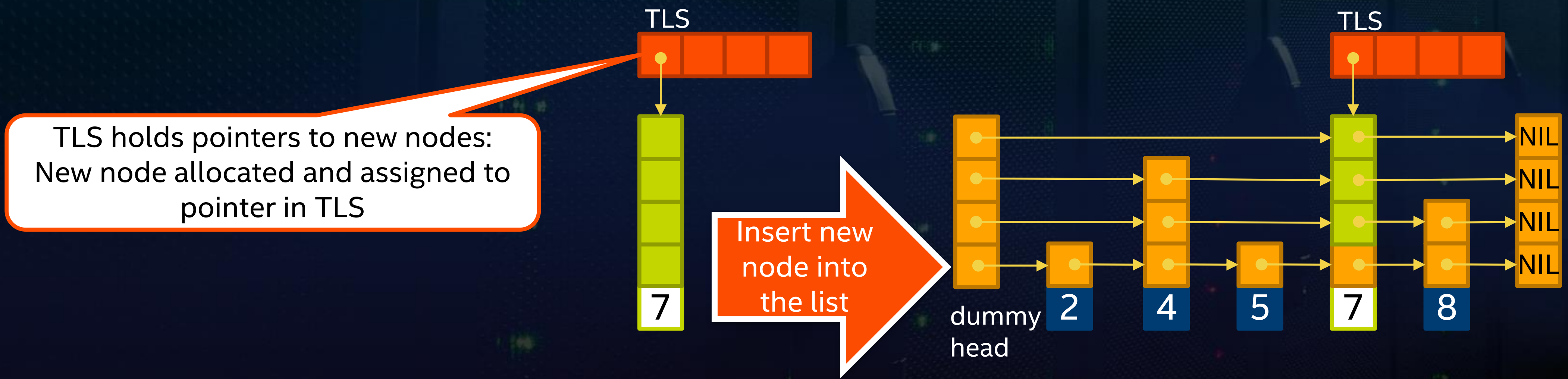
DATA CONSISTENCY IN CONCURRENT LIST-LIKE DATA STRUCTURES

```
persistent_ptr ptr; // ptr resides on-stack

{
    manual tx;
    ptr = make_persistent(...);
    commit();
}

// Memory leak, if a crash happens here

atomic_store(list>next, ptr);
persist(list->next);
```

- All nodes must be reachable after restart

DATA CONSISTENCY IN CONCURRENT LIST-LIKE DATA STRUCTURES

- Data consistency = each node is reachable after crash
- Use persistent TLS to track persistent allocations
- Each new node is always reachable via TLS
  - In case of a crash, we can redo insert if it was not completed



DATA CONSISTENCY IN CONCURRENT LIST-LIKE DATA STRUCTURES

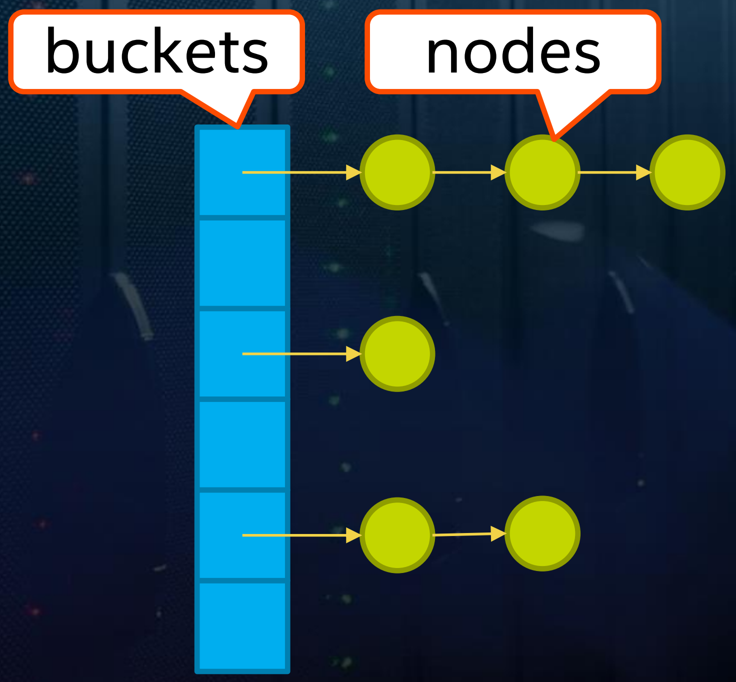
```
auto &ptls = persistent_tls.local();

{
    manual tx;
    ptls.ptr = make_persistent();
    commit();
}

atomic_store(list>next, ptls.ptr);
persist(list->next);
```

- Use Persistent TLS to avoid memory leak
- On restart, all not inserted nodes are reachable through Persistent TLS

CMAP: CONCURRENT HASH MAP

- Optimistic per-bucket Read-Write lock
  - Find() acquires read lock
  - Insert() and Erase() acquires write lock
- Each operations do following actions:
  - Finds required bucket using the hash
  - Lock the bucket for read or write access
  - Isolation: only a single writing thread can modify bucket at a time
  - Works with the nodes inside bucket

Algorithm	Average	Worst
Space	O(n)	O(n)
Search	O(1)	O(n)
Insert	O(1)	O(n)
Delete	O(1)	O(n)



LOCKS ON PMEM

```
struct hash_map_node {
    ...
    /** Next node in chain. */
    node_ptr_t next;

    /** Mutex (wrapper around pthread_mutex_t) */
    mutex_t mutex;

    /** Item stored in node */
    value_type item;
};
```

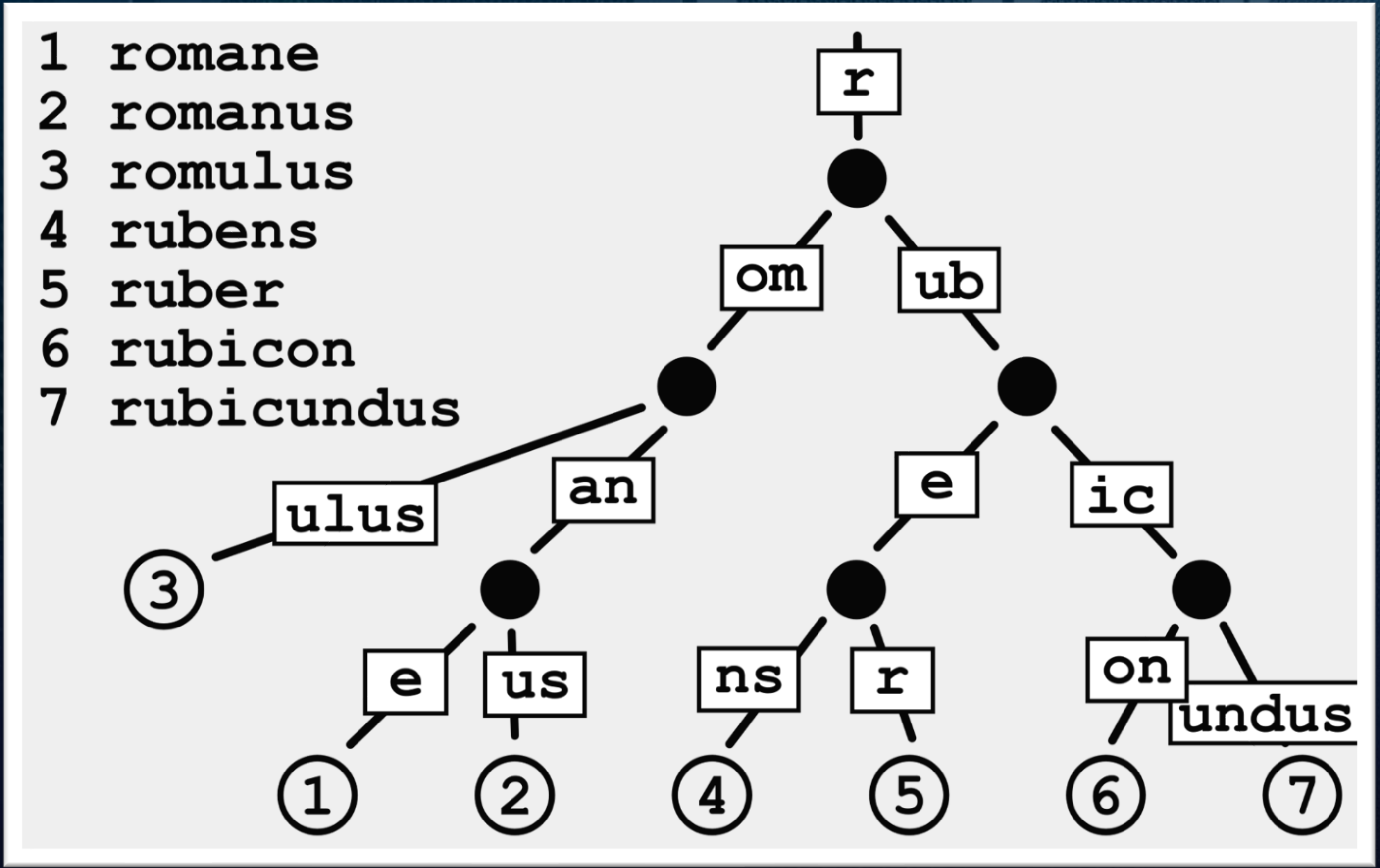
- Locking/unlocking = writing to pmem
- No explicit flush but cache lines are invalidated when accessed from different cores
- Affects cmap, csmap, vcmap

MOVING LOCKS TO DRAM

- Use sharding instead of per-element lock
- Keep per-element locks in DRAM (pointer to lock on PMEM)
  - Feasible for large elements
  - Possible with pmem::obj::concurrent_hash_map and tbb::concurrent_hash_map

```
template <typename Key, typename T,
          typename HashCompare = d1::tbb_hash_compare<Key>,
          typename Allocator = tbb_allocator<std::pair<const Key, T>>
#if __TBB_PREVIEW_CONCURRENT_HASH_MAP_EXTENSIONS
        , typename MutexType = spin_rw_mutex
#endif
         >
class concurrent_hash_map
```

RADIX TREE – BRIEF DESCRIPTION



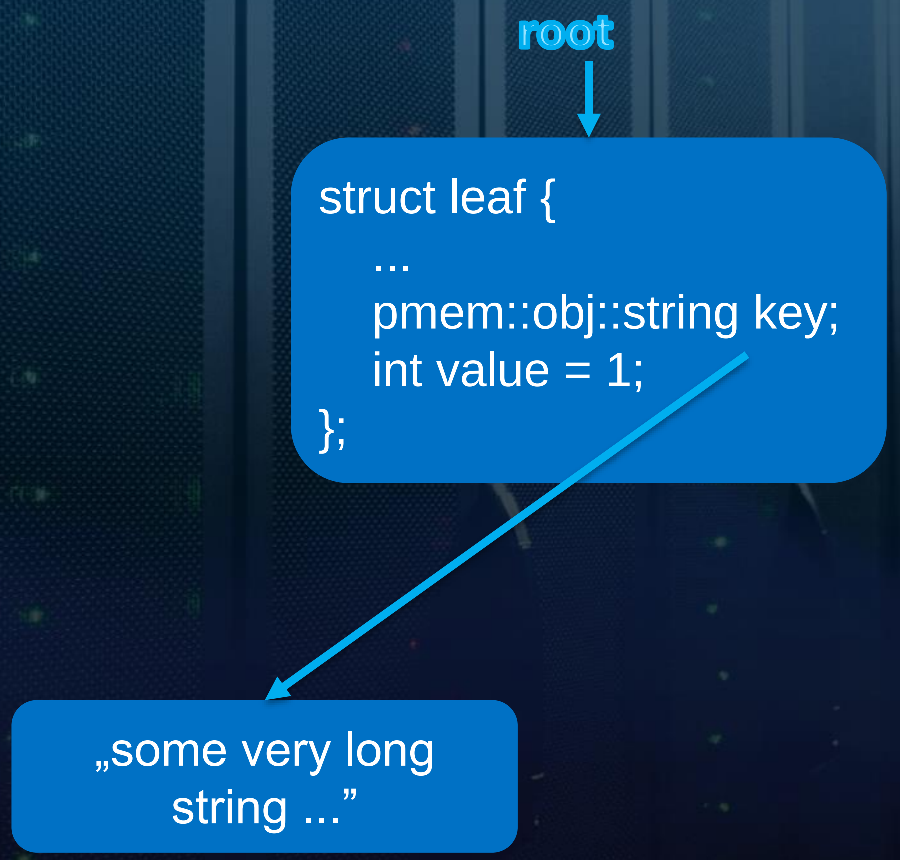
https://en.wikipedia.org/wiki/Radix_tree#/media/File:Patricia_trie.svg

RADIX TREE

- Persistent, single threaded, sorted engine
- Implementation avaialable in libpmemobj-cpp as container with std::map compatible API
- No key comparisons (less reads from pmem)
- No costly rebalancing
- Supports inline_string

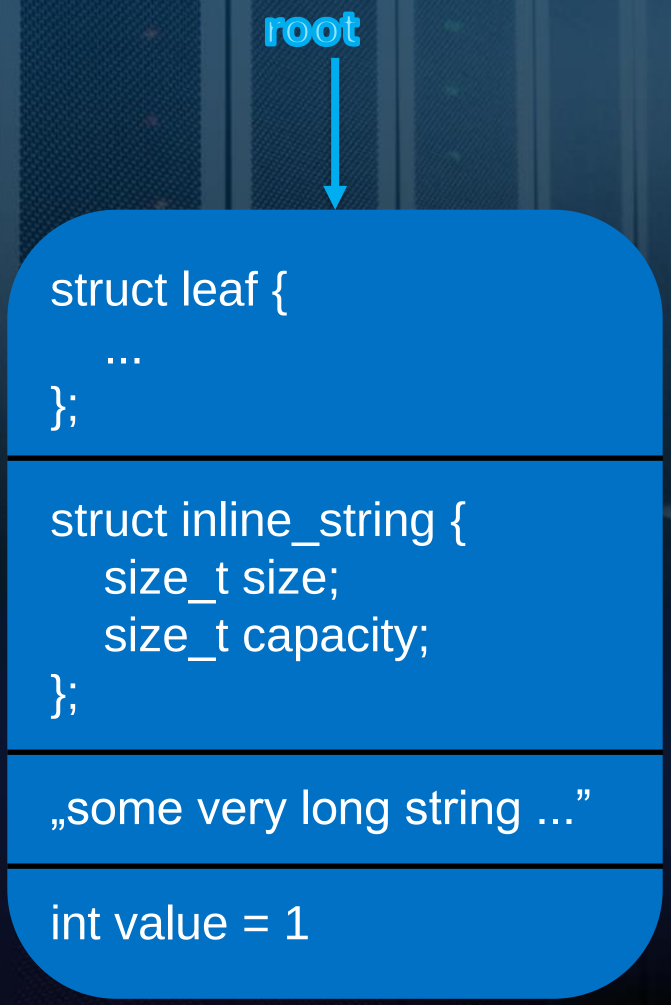
REDUCING NUMBER OF ALLOCATIONS WITH INLINE_STRING

```
radix_tree<pmem::obj::string, int> r;

...

r.try_emplace("some very long string ...",
              1); // two allocations
```



REDUCING NUMBER OF ALLOCATIONS WITH INLINE_STRING

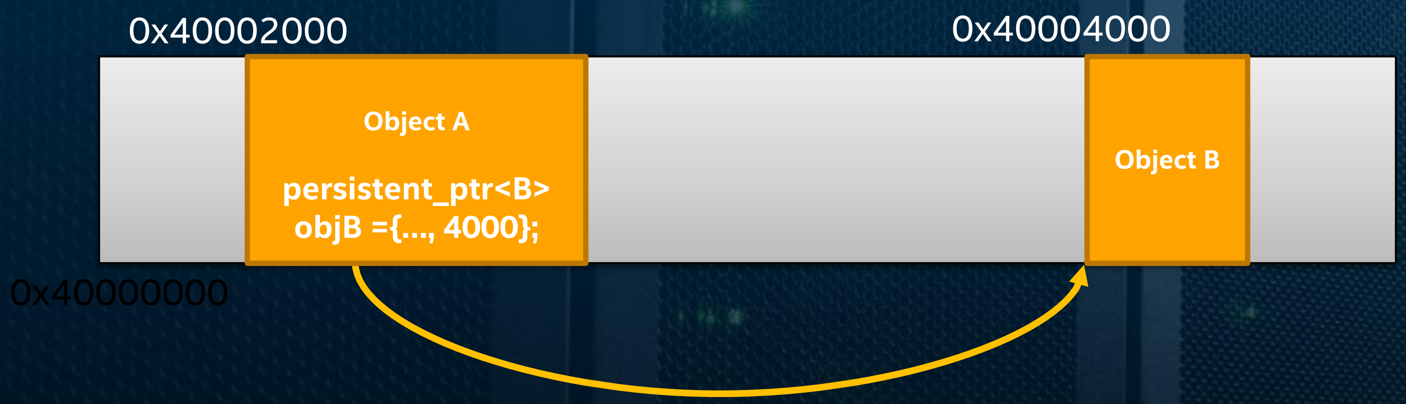
```
radix_tree<pmem::obj::inline_string, int> r;

...

r.try_emplace("some very long string ...",
              1); // one allocation
```



ROBINHOOD

- Persistent, concurrent, unsorted engine
- Supports 8B keys and values only
- Uses robinhood hashing: variant of open addressing
- Cache friendly and memory efficient
- Concurrency achieved through sharding

PMEM::OBJ::PERSISTENT_PTR PERFORMANCE PROBLEM

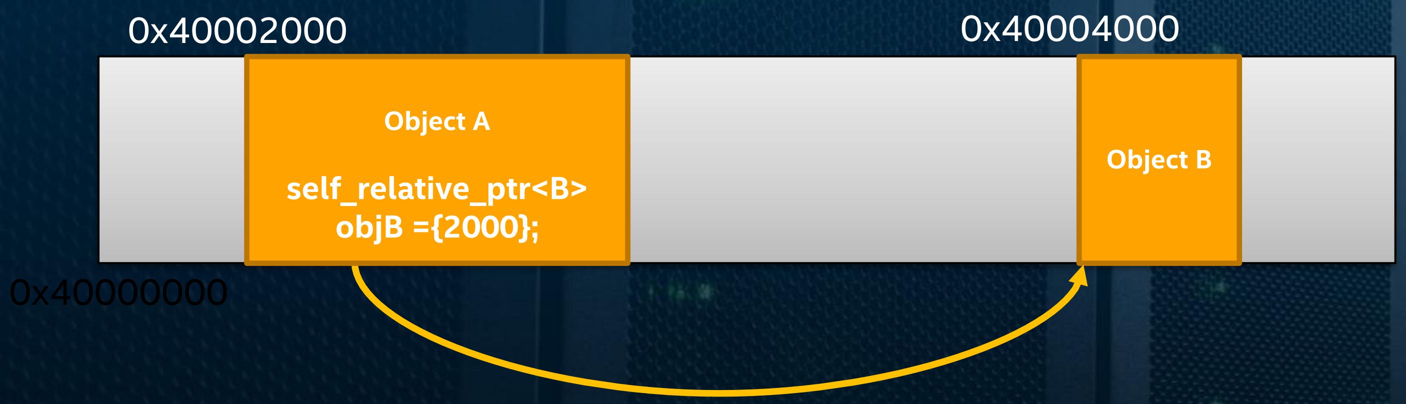



```
typedef struct pmemoid {
   uint64_t pool_uuid_lo;
   uint64_t off;
} PMEMoid;

void* pmemobj_direct(PMEMoid oid) {
    if (cache->uuid_lo != oid.pool_uuid_lo) {
        cache->pop = pmemobj_pool_by_oid(oid);
        cache->uuid_lo = oid.pool_uuid_lo;
    }
    return (void *)((uintptr_t)cache->pop + oid.off);
}
```

Cached in TLS

- Compiler cannot optimize access to cache->pop
- Each pointer derefence goes through tls

PMEM::OBJ::SELF_RELATIVE_PTR



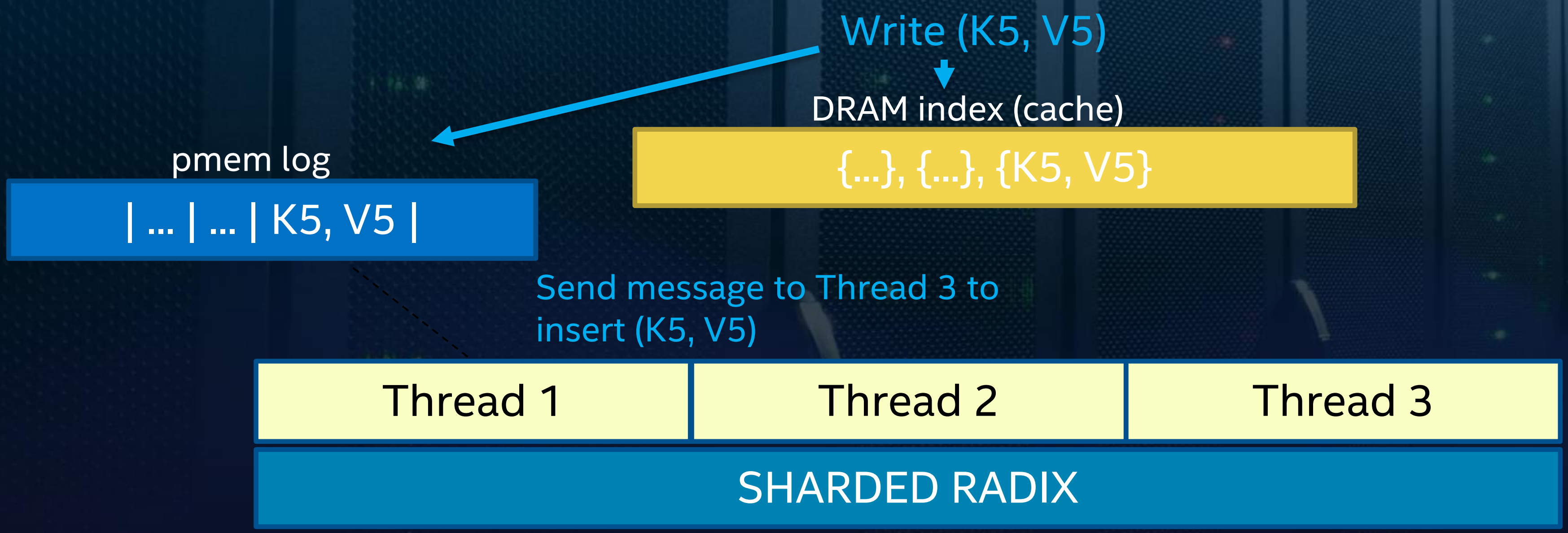
- No caching needed
- Size of self_relative_ptr is 8B
- Provides std::atomic specialization

```
T* self_relative_ptr::get() {
    if (is_null())
        return nullptr;
    return reinterpret_cast<byte_ptr_type>(const_cast<this_type *>(this)) + offset + 1;
}
```

Future plans

FUTURE PLANS

- Combining radix with a DRAM caching layer



FUTURE PLANS

- Allowing single writer and multiple readers (lock-free) for radix
  - Epoch-based reclamation
- Extending bindings functionality
- Optimizing existing engines
- Publishing regular performance reports
- Creating more educational materials about data structure design

CALL TO ACTION

- Try our data structures
  - https://github.com/pmem/libpmemobj-cpp
- Try PMEMKV in your C/C++, Java, Python or NodeJS apps
  - https://github.com/pmem/pmemkv
- Read more about persistent memory and concurrent data structures
  - https://pmem.io/book/
- Learn more about concurrency in failure atomic data structures
  - https://www.youtube.com/watch?v=6V5LcBKhpJE&t=1659s

SPDK, PMDK, Intel® Performance Analyzers | Virtual Forum

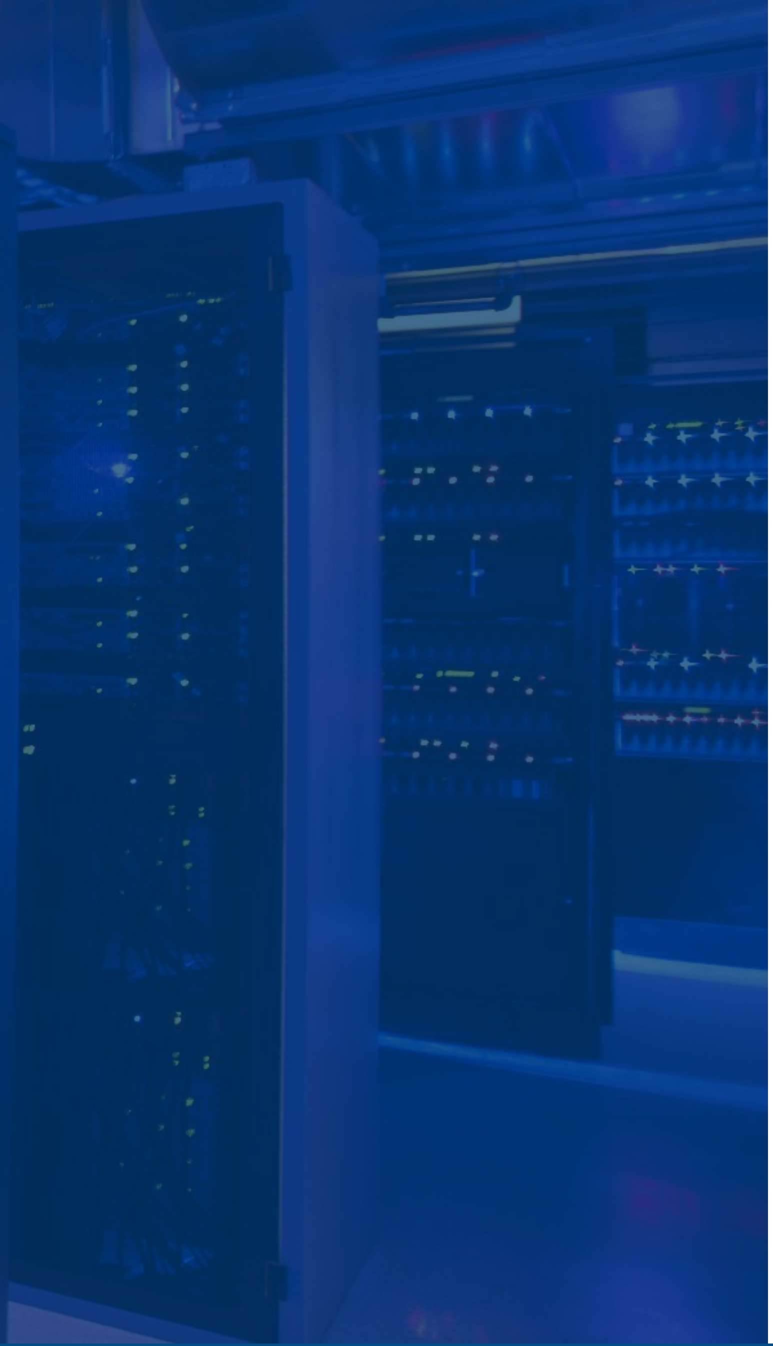
BACKUP

Presentation Title

Presenter Name

Principle Engineer
Intel

SPDK, PMDK, Intel® Performance Analyzers | Virtual Forum



Agenda

1

Sample Text 1

This is a sample text. Insert your desired text here.

2

Sample Text 2

This is a sample text. Insert your desired text here.

3

Sample Text 3

This is a sample text. Insert your desired text here.

4

Sample Text 4

This is a sample text. Insert your desired text here.

Agenda

01	Topic 1 Short description of topic
02	Topic 2 Short description of topic
03	Topic 3 Short description of topic
04	Topic 4 Short description of topic
05	Topic 5 Short description of topic
06	Topic 6 Short description of topic

IDEA 1

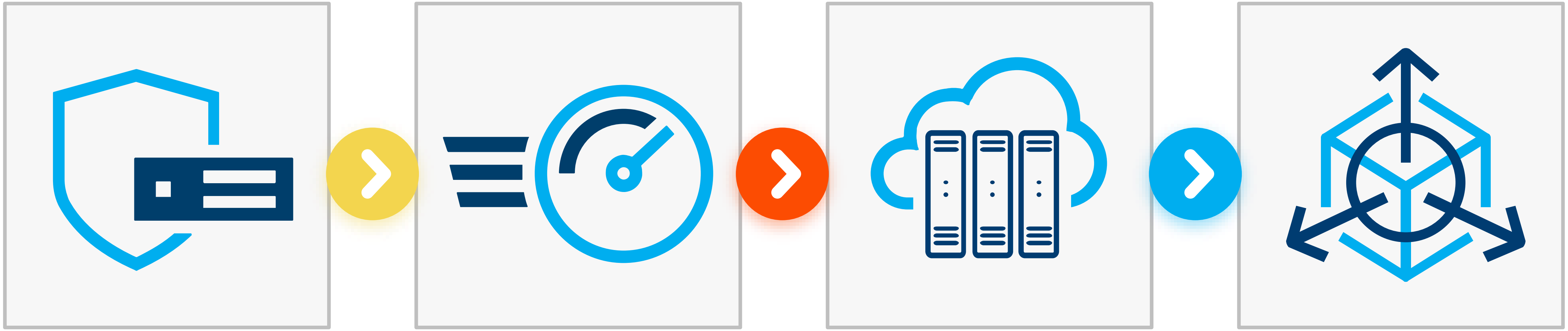
This is a sample text. Insert your desired text here. This is a sample text. Insert your desired text here.

IDEA 2

This is a sample text. Insert your desired text here. This is a sample text. Insert your desired text here.

Section Break

This is a sample text. Insert your desired text here.



This is a sample text. Enter your text here

This is a sample text. Enter your text here

This is a sample text. Enter your text here

This is a sample text. Enter your text here

This is a sample text. Enter your text here

This is a sample text. Enter your text here

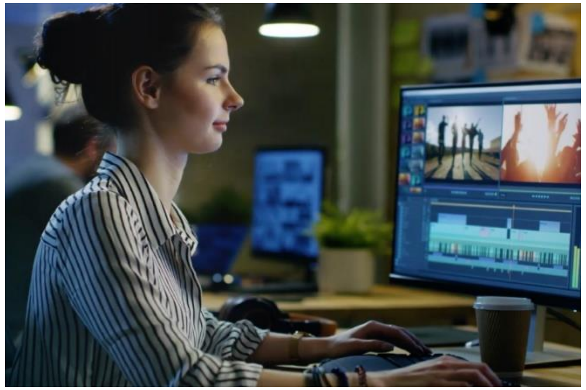
This is a sample text. Enter your text here

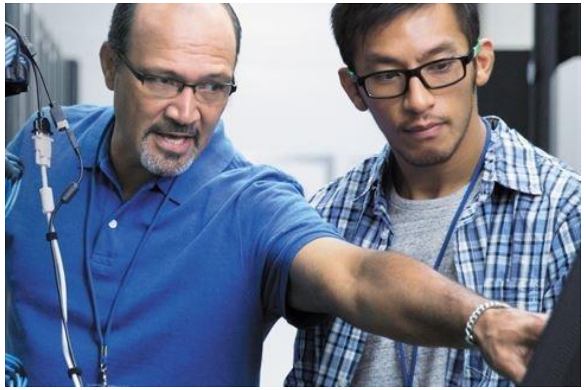
Enter Title

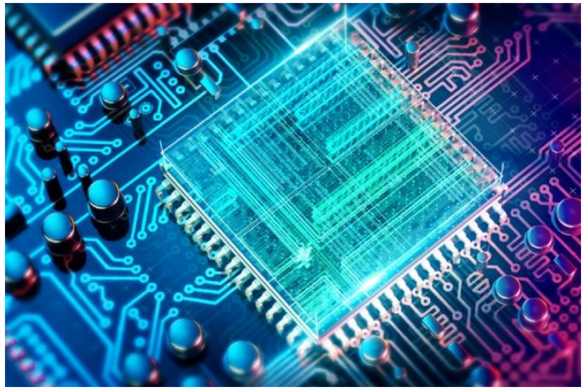
This is a sample text. Enter your text here. This is a sample text. Enter your text here. This is a sample text. Enter your text here. This is a sample text. Enter your text here.

This is a sample text. Enter your text here.
This is a sample text. Enter your text here.


APP APP APP
OS OS OS
VM






PMEMKV IS SIMPLE!

C++ example – Direct data access via Iterators – read iterator

```
result<pmem::kv::read_iterator> res_it = kv->new_read_iterator();
ASSERT(res_it.is_ok());
read_iterator &it = res_it.get_value();

auto s = it.seek_to_first();
do {
    /* read key */
    pmem::kv::result<string_view> key_result = it.key();
    assert(key_result.is_ok());

    std::cout << "key: " << key_result.get_value().data() << std::endl;
    /* read a value */
    pmem::kv::result<string_view> val_result = it.read_range();
    assert(val_result.is_ok());
    std::cout << "value: " <<  val_result.get_value().data() << std::endl;
} while (it.next() == status::OK);
```

PERFORMANCE

PMEMKV-JAVA IMPROVEMENTS