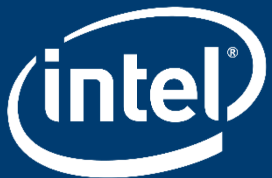


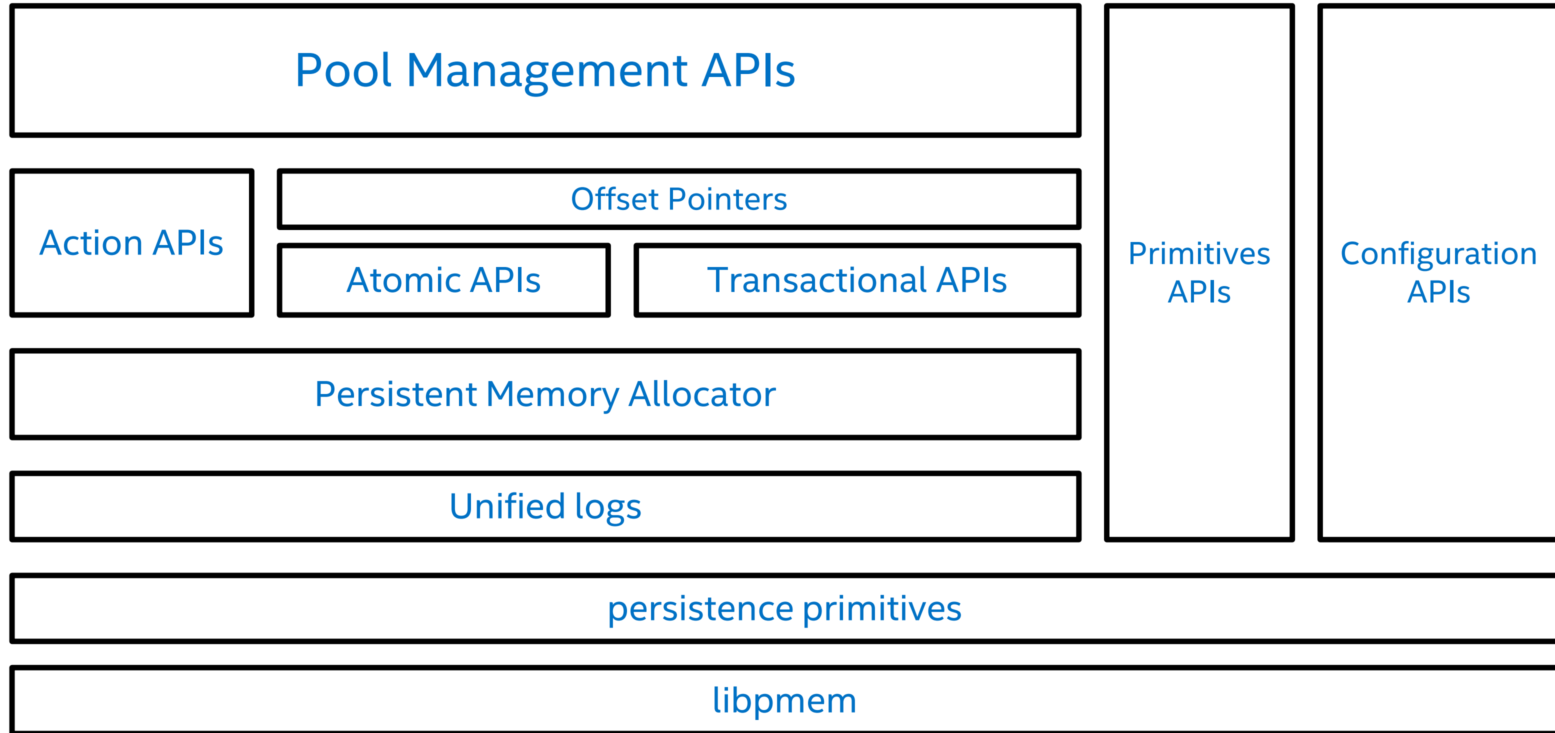
LIBPMEMOBJ & C++ API

Piotr Balcer
<piotr.balcer@intel.com>
Intel® Data Center Group

AGENDA

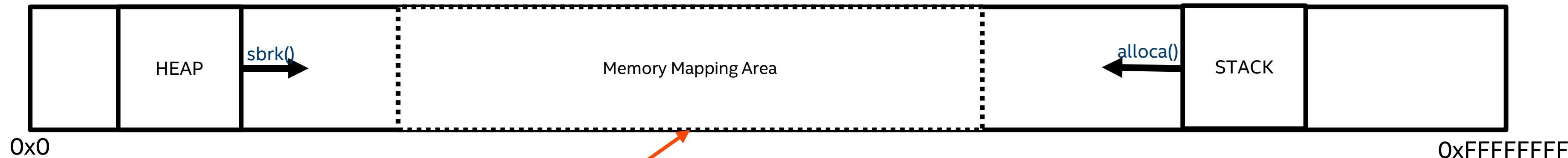
- pool<> class
- transactions
- p<> property
- peristent pointer
- transactional allocation
- persistent memory synchronization
- persistent container - array
- persistent container - vector

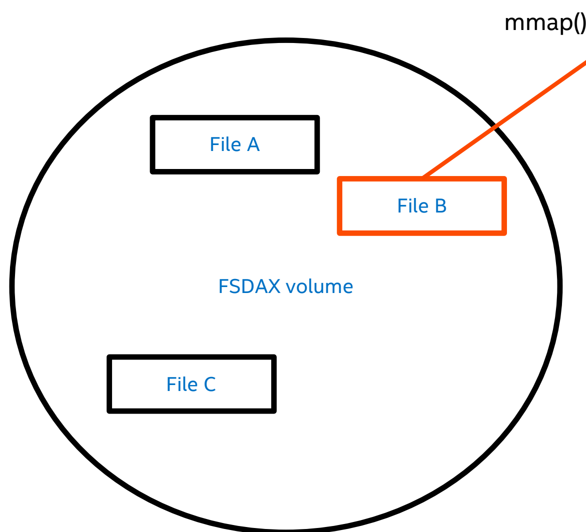
LIBPMEMOBJ OVERVIEW



intel

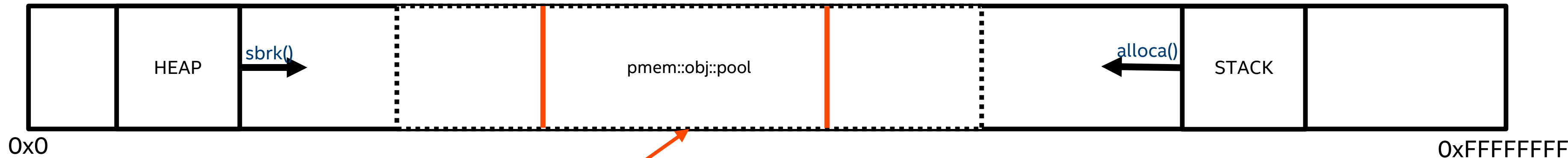
POOL MANAGEMENT API



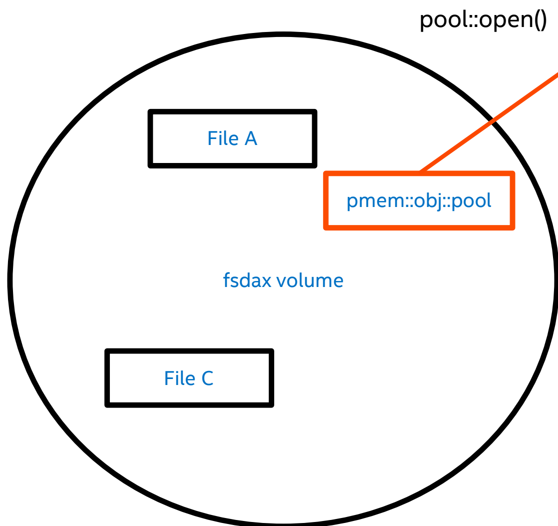


- Persistent Memory is usually exposed by the OS through a DAX-enabled file system.
- Memory Mapping is used to take advantage of byte-addressability of PMEM
- mmap() does **not** guarantee the address of the mapping. Especially if Address Space Layout Randomization (ASLR) is enabled.

POOL MANAGEMENT APIS



pool::open()



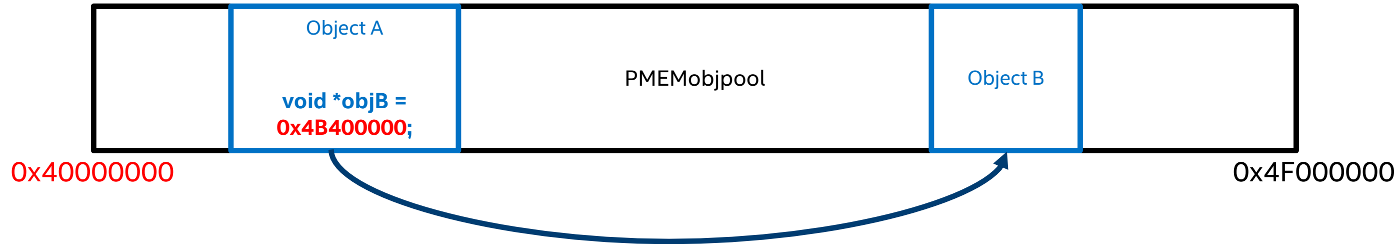
- libpmemobj abstracts away the underlying storage, providing unified APIs for managing files
- The entire library adapts to what type of storage is being used, and does the right thing for correctness.
  - This means msync() when DAX is not supported.
- It also works seamlessly for devdax devices

PMEM::OBJ::POOL EXAMPLE

```
if (access(path.c_str(), F_OK) != 0) {
    pop = pool<root>::create(path, "some_layout", PMEMOBJ_MIN_POOL,
                                   S_IRWXU);
} else {
    pop = pool<root>::open(path, "some_layout");
}
```

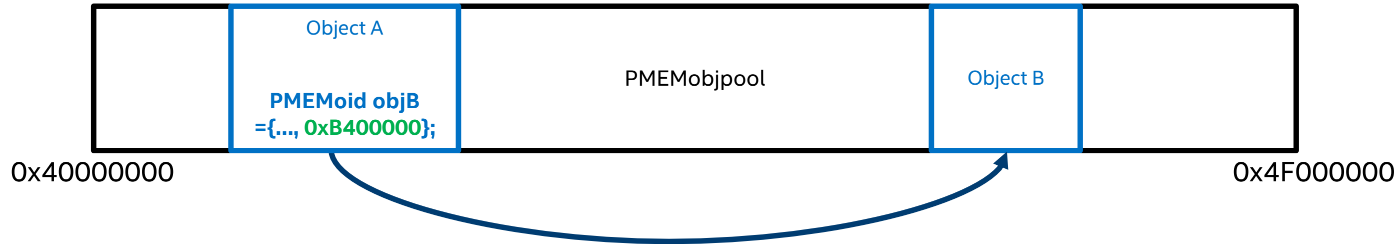
PERSISTENT POINTER

PMEM::OBJ::PERSISTENT_PTR



- The base pointer of the mapping can change between application instances
- This means that any raw pointers between two memory locations can become invalid
- Must either fix all the pointers at the start of the application
    - Potentially terabytes of data to go through…
- Or use a custom data structure which isn't relative to the base pointer

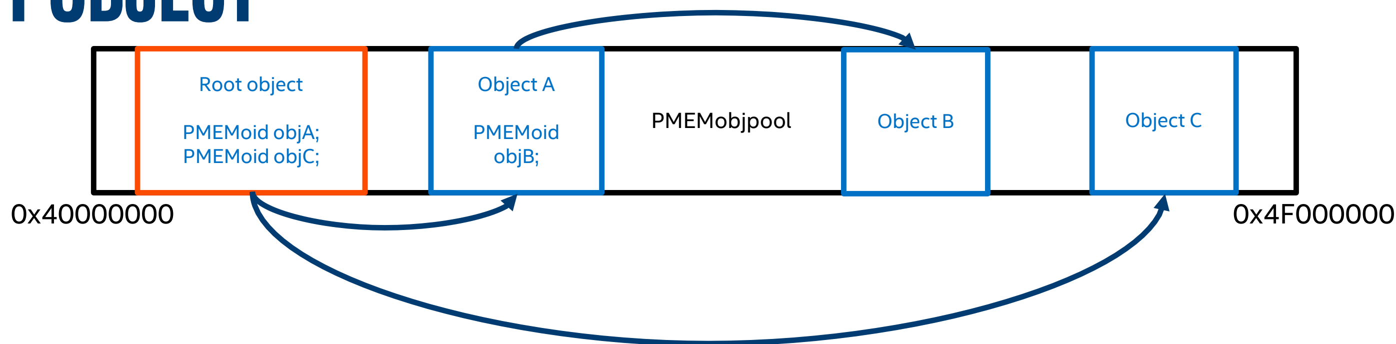
PMEM::OBJ::PERSISTENT_PTR



- libpmemobj provides 16 byte offset pointers, which contain an offset relative to the beginning of the mapping.
- Is a random access iterator
- Has primitives for flushing contents to persistence
- Does not manage object lifetime
- Does not automatically add contents to the transaction
- But it does add itself to the transaction

http://pmem.io/pmdk/manpages/linux/master/libpmemobj/oid_is_null.3

ROOT OBJECT



- All data structures of an application start at the root object.
- Has user-defined size, always exists and is initially zeroed.

- Applications should make sure that all objects are always reachable through some path that starts at the root object.
- Unreachable objects are effectively persistent memory leaks.

PMEM::OBJ::PERSISTENT_PTR EXAMPLE

```
struct foo {
    persistent_ptr<bar> barp;
};


pop = pool<foo>::create(…);
persistent_ptr<data> r = pop.root();
assert(r->barp == nullptr);
```

TRANSACTIONS

TRANSACTIONAL API

- libpmemobj provides ACID (Atomicity, Consistency, Isolation, Durability) transactions for persistent memory
  - Atomicity means that a transaction either succeeds or fails completely
  - Consistency means that the transaction transforms PMEMobjpool from one consistent state to another. This means that a pool won't get corrupted by a transaction.
  - Isolation means that transactions can be executed as if the operations were executed serially on the pool. This is optional, and requires user-provided locks.
  - Durability means that once a transaction is committed, it remains committed even in the case of system failures

TRANSACTIONS EXAMPLE

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");

transaction::run(pop, [] {

    // do some work...

}, persistent_mtx, persistent_shmtx);
```

CLOSURE TRANSACTIONS

- Take an std::function object as transaction body
- No explicit transaction commit
- Available with every C++11 compliant compiler
- Throw an exception when the transaction is aborted
- Take an arbitrary number of locks

PMEM::OBJ::P<>

PMEM::OBJ::P

- Overloads operator= for snapshotting in a transaction
- Overloads a bunch of other operators for seamless integration
  - Arithmetic
  - Logical
- Should be used for fundamental types
- No convenient way to access members of aggregate types
- No operator. to overload

CODE WITH MANUAL SNAPSHOTTING

```
struct data {
    int x;
}

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

transaction::run(pop, [&]{
    pmemobj_tx_add_range(root, 0, sizeof (struct data));
    datap->x = 5;
});
```

CODE WITH PMEM::OBJ:P

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

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

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

PMEM::OBJ::MAKE_PERSISTENT

TRANSACTIONAL ALLOCATION

- Can be used only within transactions
- Use transaction logic to enable allocation/delete rollback of persistent state
- make_persistent calls appropriate constructor
  - Syntax similar to std::make_shared
- delete_persistent calls the destructor
  - Not similar to anything found in std

TRANSACTIONAL ALLOCATION EXAMPLE

```
struct data {
    data(int a, int b) : a(a), b(b) {}
    int a;
    int b;
}

transaction::run(pop, [&]{
    persistent_ptr<data> ptr = make_persistent<data>(1, 2);
    assert(ptr->a == 1);
    assert(ptr->b == 2);

    persistent_ptr<data> ptr2 = make_persistent<data>(allocation_flag::no_flush(),
                                                      2, 3);

    ...

    delete_persistent<data>(ptr);
});
```

ALLOCATION FLAGS

- class_id(id)
  - Allocate the object from the allocation class with id equal to id
- no_flush()
  - Skip flush on commit

THREAD SYNCHRONIZATION

PERSISTENT MEMORY SYNCHRONIZATION PRIMITIVES

- Types:
  - mutex
  - shared_mutex
  - timed_mutex
  - condition_variable
- All with an interface similar to their std counterparts
- Auto reinitializing
- Can be used with transactions

PERSISTENT MEMORY CONTAINERS

PMEM::OBJ::EXPERIMENTAL::ARRAY

- std::array compatible interface (almost)
- Takes care of adding elements to a transaction
  - In operator[]/at() when obtainig non-const reference
  - On iterator dereference
  - In other methods which allow write access to data
- Works with std algorithms

PMEM::OBJ::EXPERIMENTAL::ARRAY EXAMPLE

```
transaction::run(pop, [&]{
    auto ptr = make_persistent<array<int, 6>>();

    // iterators will snapshot on element access
    std::fill(ptr->begin(), ptr->end(), 1);

    // modify all elements in a range
    for (auto &e : ptr->range(0, 3)) {
        e++;
    }

    delete_persistent<array<int, 6>>(ptr);
});
```

PMEM::OBJ::EXPERIMENTAL::VECTOR

- std::vector compatible interface (almost)
- Takes care of adding elements to a transaction
  - The same way as in array
- All functions which may alter vector properties are atomic
  - This includes: resize(), reserve(), push_back() and others
  - Transactions are used internally
  - Strong exception gurantee

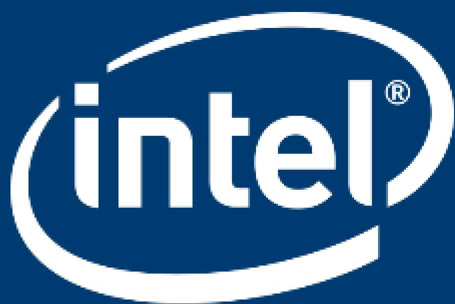
PMEM::OBJ::EXPERIMENTAL::VECTOR EXAMPLE

```
transaction::run(pop, [&]{
    auto ptr = make_persistent<vector<int>>();

    ptr->push_back(1);

    ptr->resize(10);
    ptr->at(5) = 10;

    delete_persistent<vector<int>>(ptr);
});
```


intel®