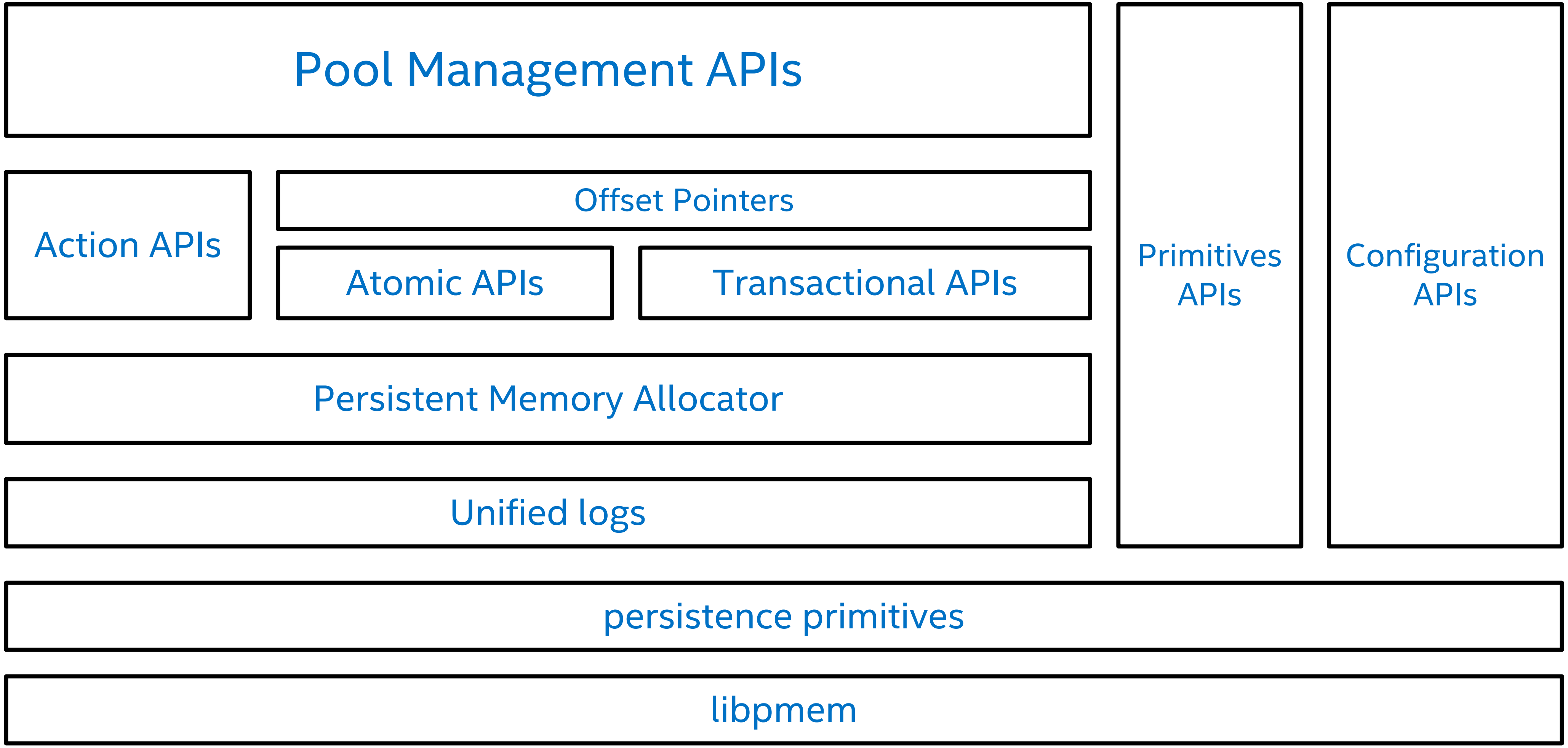
LIBPMEMOBJ API DEEP DIVE

Speaker: Szymon Romik (Intel Data Center Group)

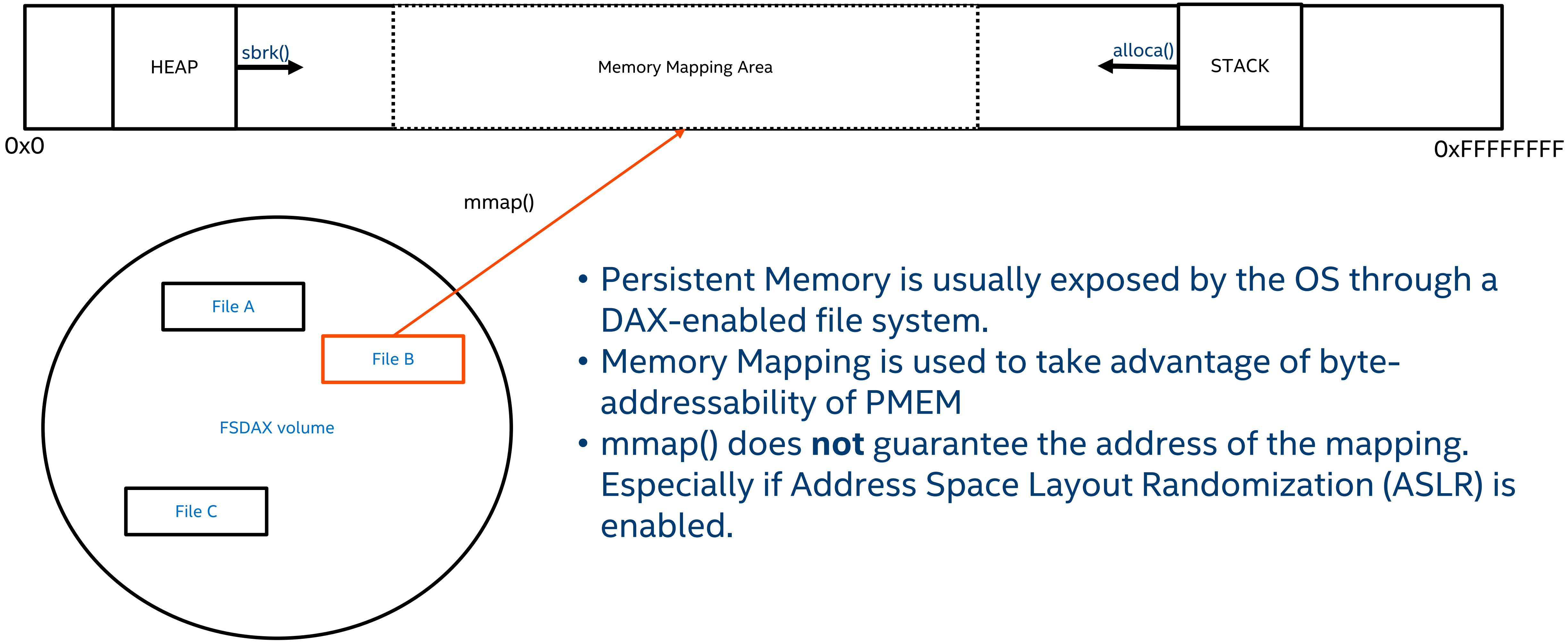
<szymon.romik@intel.com>

May, 2019

libpmemobj overview

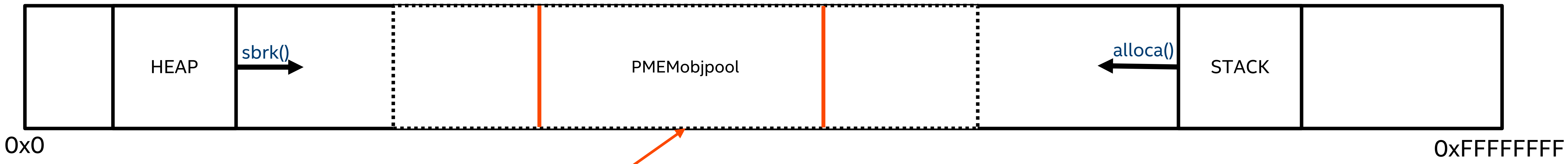


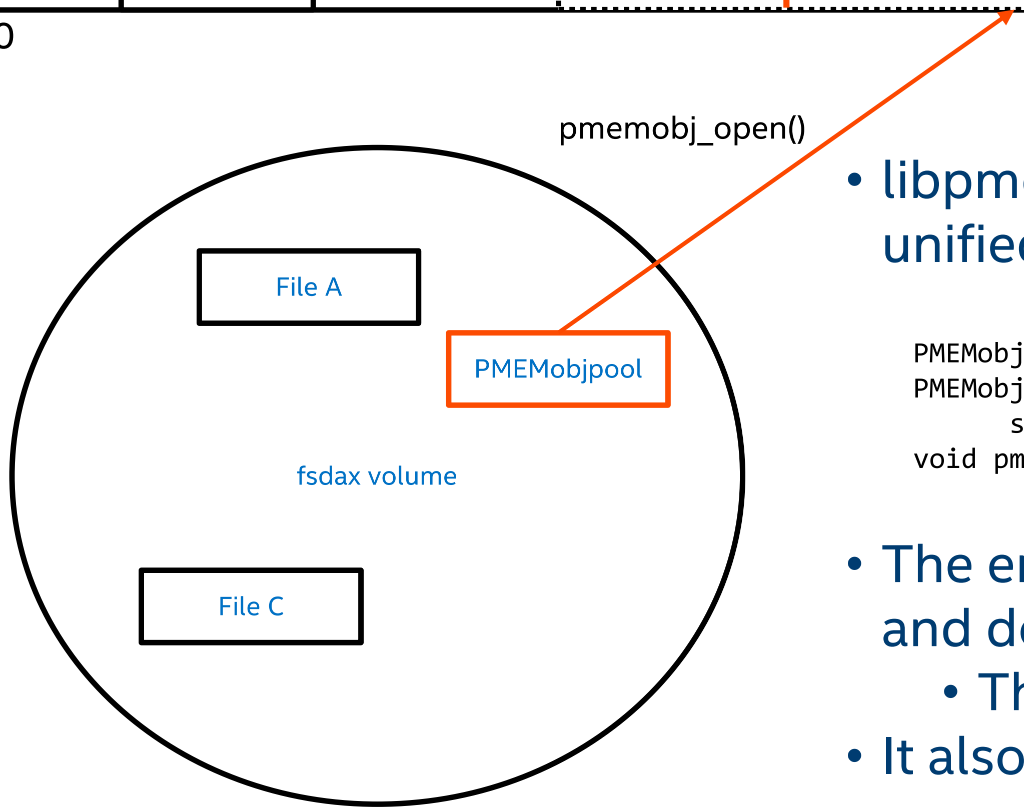
Pool Management APIs



- 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





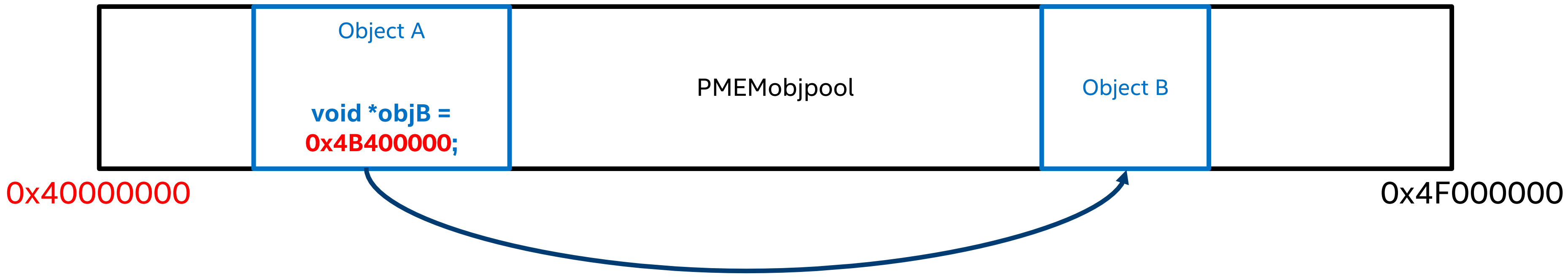
- libpmemobj abstracts away the underlying storage, providing unified APIs for managing files

```
PMEMobjpool *pmemobj_open(const char *path, const char *layout);
PMEMobjpool *pmemobj_create(const char *path, const char *layout,
      size_t poolsize, mode_t mode);
void pmemobj_close(PMEMobjpool *pop);
```

- 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

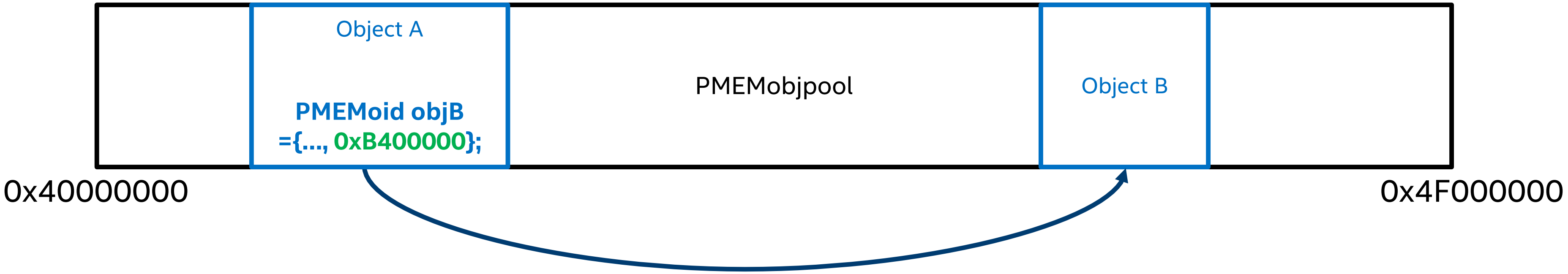
http://pmem.io/pmdk/manpages/linux/master/libpmemobj/pmemobj_open.3

Offset pointers



- 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

Offset pointers



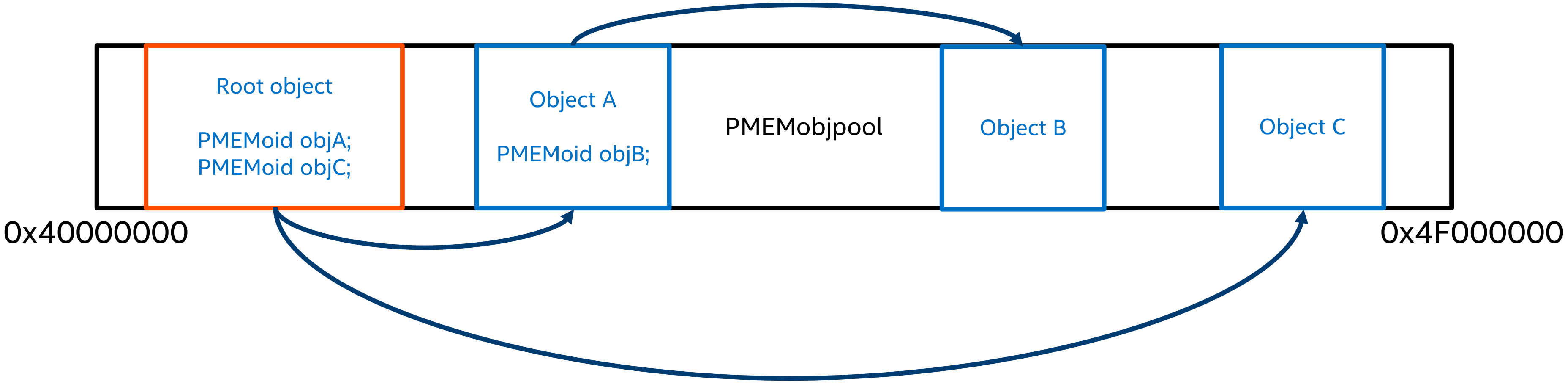
- libpmemobj provides 16 byte offset pointers, which contain an offset relative to the beginning of the mapping.
- There are also macros that add type-safety on top of the offset pointers, making their use relatively easy.

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

```
void *pmemobj_direct(PMEMoid oid);
PMEMoid pmemobj_oid(const void *addr);
```

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.

```
PMEMoid pmemobj_root(PMEMobjpool *pop, size_t size);
```

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

Primitives API

- Applications taking advantage of Persistent Memory must be failure atomic.
- libpmemobj takes care of that automatically if using Transactional APIs, but applications are free to do their own custom fail-safe atomic algorithms.
- To do that, use the built-in persistence primitives, which include:
  - Functions to force data into the persistence domain (flush/drain/persist)
  - PMEM optimized memory operations (memmove/memcpy/memset)

```
void pmemobj_persist(PMEMobjpool *pop, const void *addr, size_t len);
void *pmemobj_memcpy(PMEMobjpool *pop, void *dest, const void *src, size_t len,
unsigned flags);
```

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

Example I

```
#define LAYOUT_NAME "example_layout"
#define HELLO "Hello World"

struct root {
    uint64_t initialized;
    char data[20];
};
```

```
int main(int argc, char *argv[]) {
    const char path[] = "/mnt/pmem/myfile";
    PMEMobjpool *pop = pmemobj_open(path, LAYOUT_NAME);
    if (pop == NULL) return 1;

    PMEMoid root = pmemobj_root(pop, sizeof(struct root));
    struct root *rootp = pmemobj_direct(root);
    if (rootp->initialized) {
        printf("%s\n", rootp->data);
    } else {
        pmemobj_memcpy_persist(pop, rootp->data, HELLO,
    strlen(HELLO));
        rootp->initialized = 1;
        pmemobj_persist(pop, &rootp->initialized,
    sizeof(uint64_t));
    }

    pmemobj_close(pop);
}
```

Pool management APIs
Offset pointers & Root Object
Persistence primitives

Atomic APIs

Step 1

```
root->objA = pmalloc(pool, sizeof(struct objectA));
```

Step 2

- Memory allocation has at least two steps:
  1. Selection of the memory block to allocate
  2. Assignment of the resulting pointer to some destination pointer
- If the application is interrupted in between these steps
  - On DRAM, nothing happens, because all memory allocations vanish
  - On PMEM, memory is leaked, because the allocated object is unreachable

Atomic APIs

```
pmemobj_alloc(pool, &root->objA, sizeof(struct objectA),
   type_num, constr, constr_arg);
```

- In libpmemobj atomic API these two steps are merged into one. The object is fail-safe atomically allocated and assigned to the destination pointer.
- This API also introduces a type numbers and cunstructors
  - Type number is an 8 byte embedded metadata field which identifies the object in the pool. Can be used to recover data if objects become unreachable.
  - Constructors are used to initialize objects with data. Once an object is allocated, the constructor was ran successfully.

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

Example II

```c
int rect_construct(PMEMobjpool *pop, void *ptr, void *arg) {
    struct rectangle *rect = ptr;
    rect->x = 5;
    rect->y = 10;
    pmemobj_persist(pop, rect, sizeof *rect);

    return 0;
}
int area_calc(const struct rectangle *rect) {
    return rect->a * rect->b;
}
```

Atomic Allocation/Free

Constructor

```c
pmemobj_alloc(pop, &root->rect, struct rectangle,
    rect_construct, NULL);
int a = area_calc(D_RO(root)->rect);
/* busy work */
pmemobj_free(&D_RW(root)->rect);
```

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

Transactional API

REDO & UNDO logging

- These are key concepts to understand when dealing with transactions.
- libpmemobj has a unified implementation of the two, and they are used in conjunction

Transactional API

REDO logging

- Redo logs are used when immediate visibility of data is not required
- All modifications are stored in separately to the data being modified
- Once the transaction is complete, some kind of finish flag is set
    - A bit flag, checksum, etc
- If redo log is complete, application will attempt to apply it until successful

Transactional API

UNDO logging

- Each undo log entry is a snapshot of some other location in memory
- Allows modifications to be done in-place once the log entry is created
- Once the transaction is complete, the log is discarded
- Otherwise, in case of an abort, the log entries are applied

Transactional API

- Inside of a transaction the application can:
  - Allocate new objects
  - Free existing objects
  - Modify existing objects
  - Isolate objects

```
TX_BEGIN_PARAM(pool, TX_PARAM_MUTEX, &root->lock, TX_PARAM_NONE) {
    pmemobj_tx_add_range_direct(root, sizeof(*root));
    root->objA = pmemobj_tx_alloc(sizeof(struct objectA), type_num);
    pmemobj_tx_free(root->objB):
    root->objB = OID_NULL;
} TX_END
```

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

Transactional heap operations

```
TX_BEGIN(pool) {
    root->objA = pmemobj_tx_alloc(sizeof(struct objectA), type_num);
} TX_END
```

- Normal two step allocation is possible inside of a transaction
- All metadata modifications of heap operations inside of a single transactions are aggregated.
    - This means that it's better to allocate/free many objects inside of a single transaction

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

Transactional memory modifications

```
TX_BEGIN(pool) {
    pmemobj_tx_add_range_direct(&root->value, sizeof(root->value));
    root->value = 123;
} TX_END
```

- The C API requires that all memory modifications inside of a transaction must be instrumented with “add_range” functions.
  - They create the snapshots of data for the UNDO log
  - Only needed for existing memory
- Snapshots have 24 bytes of metadata

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

Type safety macros

- PMEMoid, our persistent pointer, is just a data structure.
  - It carries no type information.
  - This makes all PMEMoid equivalent to raw `void *` pointers.
- To enable applications to regain some type safety, we've developed a set of macros that associate a persistent pointer with object's type.

```
struct btree_node {
    int64_t key;
    TOID(struct btree_node) slots[2];
    char value[];
};

struct btree {
    TOID(struct btree_node) root;
};
```

```
POBJ_LAYOUT_BEGIN(btree);
POBJ_LAYOUT_ROOT(btree, struct btree);
POBJ_LAYOUT_TOID(btree, struct btree_node);
POBJ_LAYOUT_END(btree);
```

```
TOID(struct btree) btree = POBJ_ROOT(pop, struct btree);
TOID(struct btree_node) node = D_RO(btree)->root;
D_RW(node)->key = 1234;
```

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

Example III

```
TOID(struct my_root) root = POBJ_ROOT(pop);
TX_BEGIN(pop) {
    TX_ADD(root); /* we are going to operate on the root object */
    TOID(struct rectangle) rect = TX_NEW(struct rectangle);
    D_RW(rect)->x = 5;
    D_RW(rect)->y = 10;
    D_RW(root)->rect = rect;
} TX_END

int p = area_calc(D_RO(root)->rect);
/* busy work */
```

Transactional Allocation/Free

Transactional modification

Actions API

- Previous APIs combined memory allocation and initialization into a single atomic operation (either a function or a transaction).
- This makes it difficult to handle workloads with long pauses between the two.
  - For example, networked application which buffers data into persistent memory
- Actions API allows the application to first reserve some persistent memory buffer in volatile state, and publish it some time later.
- Objects allocated this way must be manually persisted.

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

Example IV

Reservation
Persisting
Publication
Transactional Modification

```
TOID(struct my_root) root = POBJ_ROOT(pop);

struct pobj_action action;

TOID(struct rectangle) rect = POBJ_RESERVE(pool, struct rectangle, &action);
D_RW(rect)->x = 5;
D_RW(rect)->y = 10;
pmemobj_persist(pop, D_RW(rect), sizeof(struct rectangle));

TX_BEGIN(pop) {
    pmemobj_tx_publish(&action, 1); /* move the reservation into TX */
    TX_ADD(root);
    D_RW(root)->rect = rect;
} TX_END
```

Configuration APIs

- libpmemobj contains a large variety of configuration options, all exposed through a unified interface: CTL
- It allows setting the configuration options through:
  - Files
  - Environment variables
  - Function calls

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

Example V

```
struct my_object {
    int a;
    int b;
    char data[123];
};
```

Custom allocation class for a specific data structure of an application

```
struct pobj_alloc_class_desc class;
class.header_type = POBJ_HEADER_NONE;
class.unit_size = sizeof(struct my_object);
class.alignment = 0;
class.units_per_block = 100;

pmemobj_ctl_set(pop, "heap.alloc_class.new.desc", &class);
pmemobj_xalloc(pop, &root->my, sizeof(struct my_object), 0, CLASS_ID(class.id),
    NULL, NULL);
```

Summary

- libpmemobj is a vast and powerful library with a lot of flexibility.
- This might feel daunting at first, but programmers can start with highly optimized transactional API, and transition to different approaches if needed.
- We recommend the C library for low-level language bindings and where C is the only option, otherwise we recommend using far more approachable C++ bindings.

intel®