C++ AND PERSISTENT MEMORY TECHNOLOGIES, LIKE INTEL'S 3D-XPOINT

Tomasz Kapela, @tomaszkapela

CppCon, Bellevue 2017

C++ AND PERSISTENT MEMORY TECHNOLOGIES, LIKE INTEL'S 3D-XPOINT

Tomasz Kapela, @tomaszkapela

CppCon, Bellevue 2017

THIS IS NOT ABOUT 3D-XPOINT, IT'S ABOUT C++

THE MODEL

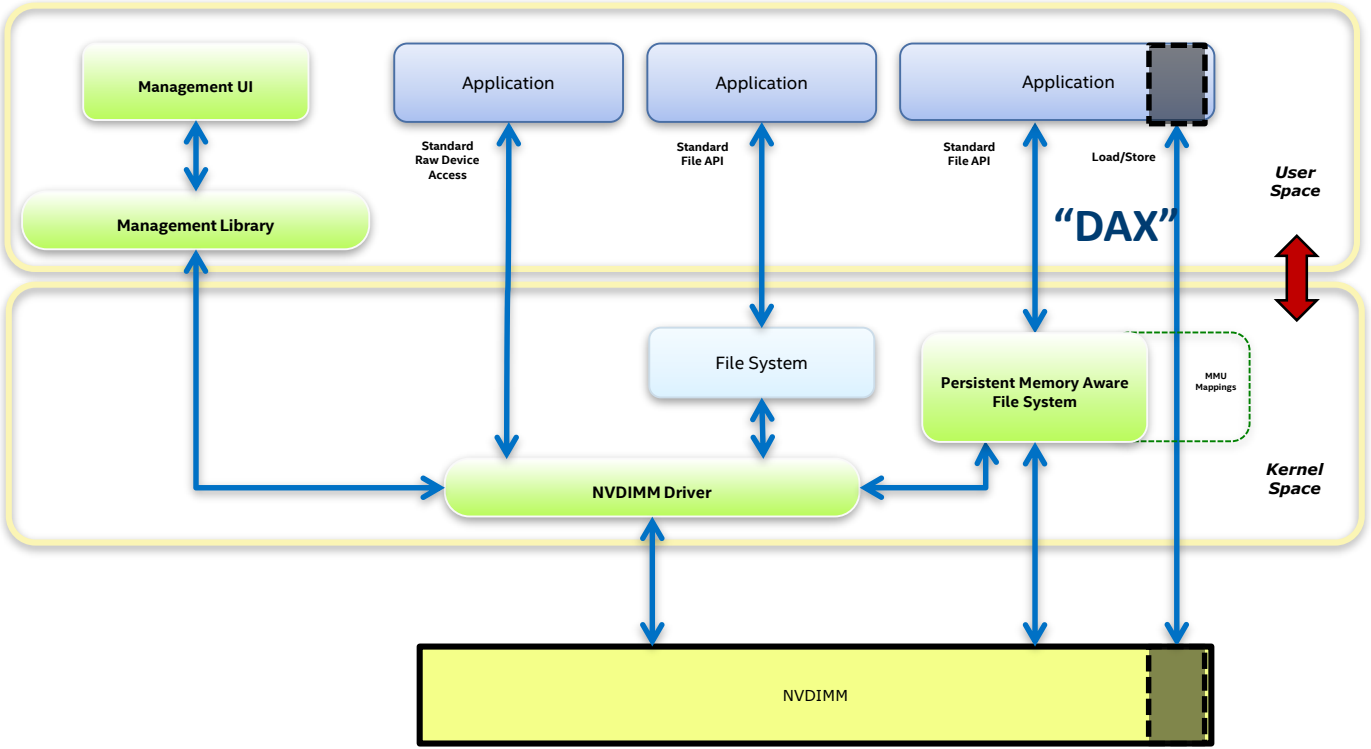
Programming Model:
At least four meanings…

1. Interface between HW and SW
2. Instruction Set Architecture (ISA)
3. Exposed to Applications (by the OS)
4. The Programmer Experience

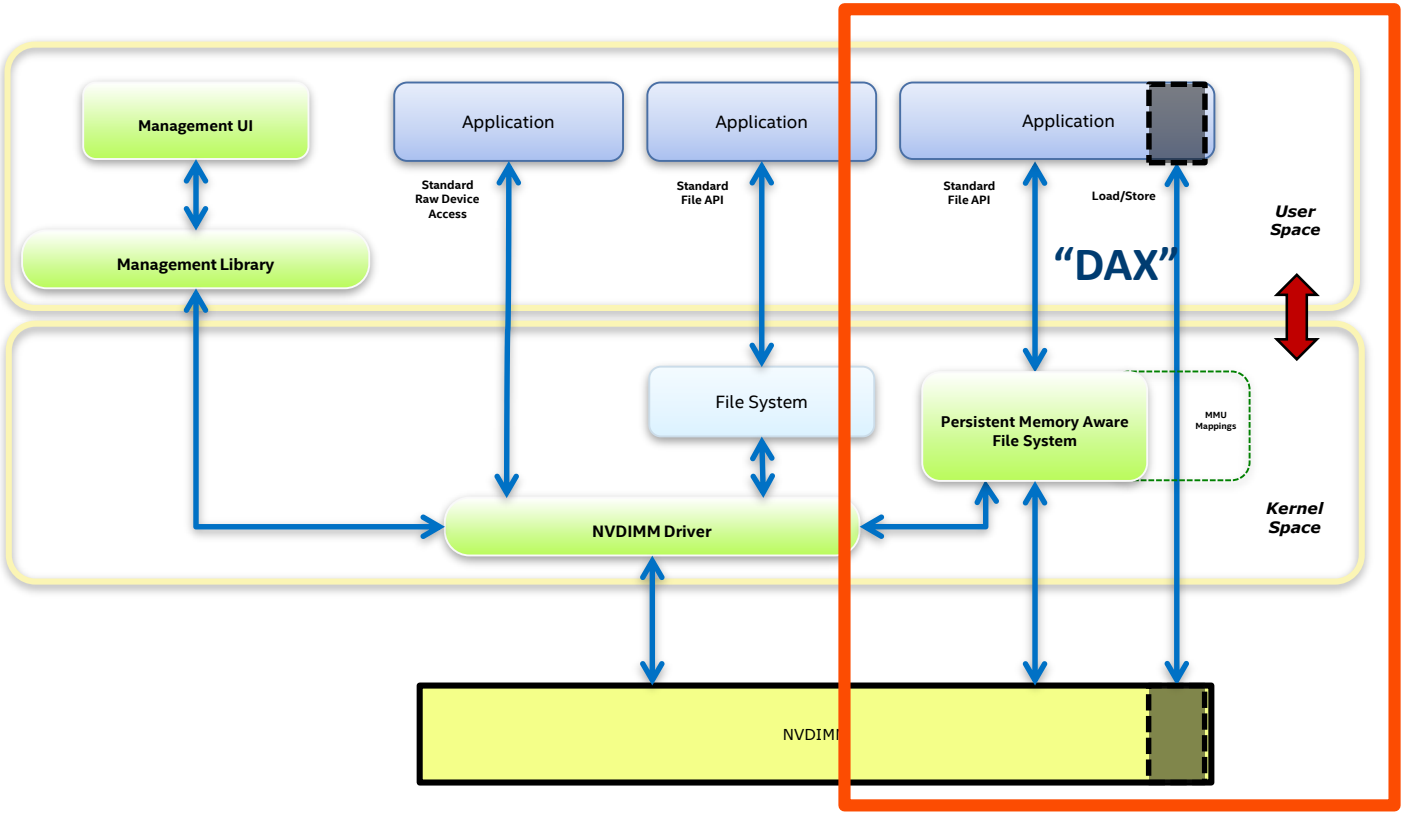
Programming Model:
At least four meanings…

1. Interface between HW and SW
2. Instruction Set Architecture (ISA)
3. Exposed to Applications (by the OS)
4. The Programmer Experience

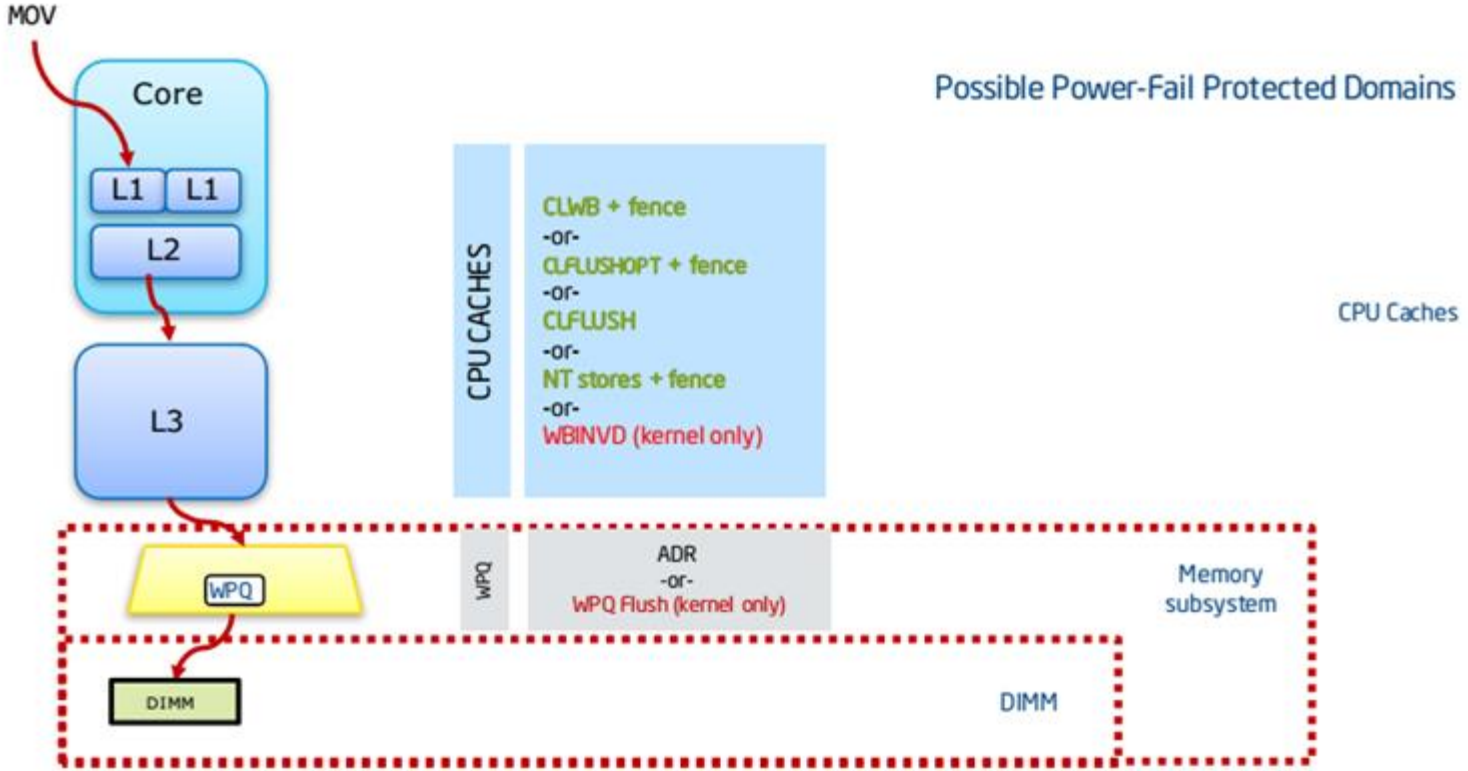
Programming Model (meaning 3): Exposing to Applications



Programming Model (meaning 3): Exposing to Applications



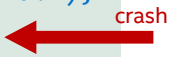
Data persistency



Atomicity

Flushing to Persistence

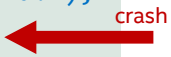
```
open(…);

mmap(…);

strcpy(pmem, „Hello, World!");

pmem_persist(pmem, 14);
```



Atomicity

Flushing to Persistence

```
open(…);

mmap(…);

strcpy(pmem, „Hello, World!");

pmem_persist(pmem, 14);
```

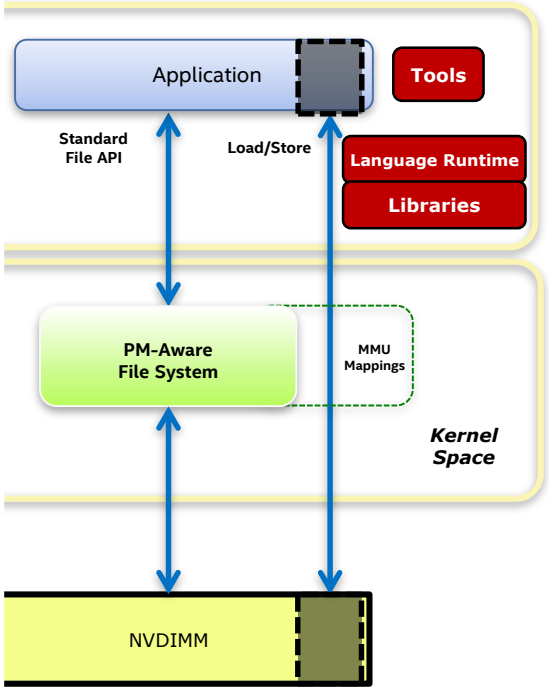


Crossing the 8-Byte Store

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”

Programming Model (meaning 4): The Programmer Experience



Result:
Safer, less error-prone, idiomatic in common languages

C++ std::vector

```
using pvector = vector<p<int>, pallocator>;
pvector persistent_vector = allocate_from_persistent_memory();

…
persistent_vector.push_back(42);
```

C++ std::vector

```
std::vector<p<int>> persistent_vector = allocate_from_persistent_memory();

…
persistent_vector.push_back(42);
```

No flush calls.
Transactional.
C++ library handles it all.

C++ std::vector

```
std::vector<p<int>> persistent_vector = allocate_from_persistent_memory();

…
persistent_vector.push_back(42);
```

No flush calls.
Transactional.
C++ library handles it all.

See “pilot” project at: `https://github.com/pmem/libcxx`

GROUND RULES

Glossary

- Persistent Memory/Storage Class Memory/Non-Volatile Memory – fast, byte addressable memory which retains contents between power cyles
- Pool – a contiguous blob of persistent memory in a process' virtual address space

LET’S START WITH A QUEUE

Queue Entry

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    int value;
};

// somewhere in the queue class

shared_ptr<entry> head;
shared_ptr<entry> tail;
```

Push

```
void push(int value)
{
    auto n = make_shared<entry>(value, nullptr);

    if (head == nullptr) {
        head = tail = n;
    } else {
        tail->next = n;
        tail = n;
    }
}
```

Pop

```
int pop()
{
    if (head == nullptr)
        throw runtime_error("Nothing to pop");

    auto ret = head->value;
    head = head->next;

    if (head == nullptr)
        tail = nullptr;

    return ret;
}
```

TRANSACTIONS

Push

```
void push(int value)
{
    transaction::exec_tx(pool, [this, &value] {
        auto n = make_shared<entry>(value, nullptr);

        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
    });
}
```

Scope of the transaction

Pop

```
int pop()
{
    transaction::exec_tx(pool, [this] {
        if (head == nullptr)
            throw runtime_error("Nothing to pop");

        auto ret = head->value;
        head = head->next;
        if (head == nullptr)
            tail = nullptr;

        return ret;
    });
}
```

Scope of the transaction

Transactions

- Undo log based transactions
- ACID like properties
- Can be nested
- In case of interruption it is rolled-back or completed upon next pool open
- Locks are held until the end of a transaction

Manual Transaction Example

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

     // do some work...

     transaction::commit();
}
```

```
auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

Pool handle

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

     // do some work...

     transaction::commit();
}



auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

Root object type

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

     // do some work...

     transaction::commit();
}



auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

Custom layout string

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

     // do some work...

     transaction::commit();
}



auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

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

     // do some work...

     transaction::commit();
}
```

Synchronization point

```
auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

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

     // do some work...

     transaction::commit();
}
```

Has to be explicit

```
auto aborted = transaction::get_last_tx_error();
```

Manual Transaction Example

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

     // do some work...

     transaction::commit();
}

auto aborted = transaction::get_last_tx_error();
```

Locks drop here

Manual Transaction Example

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

     // do some work...

     transaction::commit();
}
```

Well... This is embarrassing

```
auto aborted = transaction::get_last_tx_error();
```

Manual Transactions

- Based on the familiar RAII concept
- Fairly easy to use
  - Explicit transaction commit because of std::uncaught_exception
  - Does not throw en exception on transaction abort
- By default aborts to account for third-party exceptions or amnesia
- Accepts an arbitrary number of (persistent memory resident) locks

Automatic Transaction Example

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");
try {
     transaction::automatic(pop, persistent_mtx, persistent_shmtx);
     // do some work...
} catch (...) {
     // do something meaningful
}

auto aborted = transaction::get_last_tx_error();
```

Automatic Transaction Example

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

try {
     transacti                    op, persistent_mtx, persistent_shmtx);
     // do some work...
} catch (...) {
     // do something meaningful
}

auto aborted = transaction::get_last_tx_error();
```

No more commit

Automatic Transaction Example

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

try {
	transaction::automatic(pop, persistent_mtx, persistent_shmtx);

	...
} catch (...) {
	// do something meaningful
}

auto aborted = transaction::get_last_tx_error();
```

Locks drop here

Automatic Transaction Example

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

try {
    transaction::automatic(pop, persistent_mtx, persistent_shmtx);
    ...
} catch (...) {
    // do something meaningful and mind your locks!
}

auto aborted = transaction::get_last_tx_error();
```

Locks drop here

Automatic Transaction Example

```
auto pop = pool<root>::open("/path/to/poolfile", "layout string");
try {
     transaction::automatic(pop, persistent_mtx, persistent_shmtx);
     // do some work...
} catch (...) {
     // do something meaningful and mind your locks!
}

auto aborted = transaction::get_last_tx_error();
```

Still embarrassing?

Automatic Transactions

- Functionally and semantically identical to the manual transaction
- No explicit transaction commit
- Need C++17
  - Relies on std::uncaught_exception***s***

Closure Transaction Example

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

transaction::exec_tx(pop, [] {
     // do some work...
}, persistent_mtx, persistent_shmtx);


auto aborted = transaction::get_last_tx_error();
```

Closure Transaction Example

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

transaction::exec_tx(pop, [] {
     // do some work...
}, persistent_mtx, persistent_shmtx);


auto aborted = transaction::get_last_tx_error();
```

Awkward

Closure Transaction Example

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

transaction::exec_tx(pop, [] {
     // do some work...
}, persistent_mtx, persistent_shmtx);
```

Not embarrassing anymore

~~auto aborted = transaction::get_last_tx_error();~~

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
  - Unfortunately at the very end

Resides in persistent memory

Queue Entry Recap

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    int value;
};
```

Queue Entry Recap

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    int value;
};
```

Allocated, modified and deleted within a transaction

Queue Entry Recap

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    int value;
};
```

Needs to be snapshot

Allocated, modified and deleted within a transaction

Partially Persistent Queue Entry

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    p<int> value;
};
```

Does the actual snapshot

Allocated, modified and deleted within a transaction

The p<> Property

- AKA the workhorse
- 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

PERSISTENT POINTER

Partially Persistent Queue Entry Recap

```
/* entry in the list */
struct entry {
    shared_ptr<entry> next;
    p<int> value;
};
```

Allocated, modified and deleted within a transaction

Partially Persistent Queue Entry Recap

```
/* entry in the list
struct entry {
    shared_ptr<entry> next;
    p<int> value;
};
```

Needs to be snapshot

Allocated, modified and deleted within a transaction

Fully Persistent Queue Entry

```
/* entry in the list */
struct entry {
    persistent_ptr<entry> next;
    p<int> value;
};
```

Allocated, modified and deleted within a transaction

Fully Persistent Queue Entry

```
/* en                                    */
struct
    persistent_ptr<entry> next;
    p<int> value;
};
```

Now will be snapshot

Allocated, modified and deleted within a transaction

Persistent Smart Pointer

- Points to objects within a persistent memory pool
  - Manages translating persistent addresses to runtime addresses
- Is a random access iterator
- Has primitives for flushing contents to persistence
- std::allocator friendly

Persistent Smart Pointer

- Does not manage object lifetime
- Does not automatically add contents to the transaction
  - But it does add itself to the transaction
- Does not point to polymorphic objects
  - No good way to rebuild runtime state after pool reboot

Persistent Queue Entry

```
/* entry in the list */
struct entry {
    persistent_ptr<entry> next;
    p<int> value;
};

// somewhere in the queue class

persistent_ptr<entry> head;
persistent_ptr<entry> tail;
```

Push Recap

```
void push(int value)
{
    transaction::exec_tx(pool, [this] {
        auto n = make_shared<entry>(value, nullptr);

        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
    });
}
```

Persistent Push

```
void push(int value)
{
    transaction::exec_tx(pool, [this] {
        auto n = make_persistent<entry>(value, nullptr);

        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
    });
}
```

Persistent Push

```
void push(int value)
{
    transaction::exec_tx
        auto n = make_persistent<entry>(value, nullptr);

        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
    });
}
```

Allocation now part of the transaction

Persistent Pus

The pool handle for completeness

```
void push(pool_base &pool, int value)
{
    transaction::exec_tx(pool, [this] {
        auto n = make_persistent<entry>(value, nullptr);

        if (head == nullptr) {
            head = tail = n;
        } else {
            tail->next = n;
            tail = n;
        }
    });
}
```

Pop Recap

```
int pop() {
    transaction::exec_tx(pool, [this] {
        if (head == nullptr)
            throw runtime_error("Nothing to pop");

        auto ret = head->value;
        head = head->next;
        if (head == nullptr)
            tail = nullptr;

        return ret;
    });
}
```

Persistent Pop

```
int pop() {
    transaction::exec_tx(pool, [this] {
        auto ret = head->value;
        auto tmp_entry = head->next;

        delete_persistent<entry>(head);

        head = tmp_entry;

        if (head == nullptr)
            tail = nullptr;
        return ret;
    });
}
```

Code removed to improve readability

Persistent Pop

```
int pop() {
    transactio                    , [this] {
        auto re                   ;
        auto tmp_entry = head->next;

        delete_persistent<entry>(head);

        head = tmp_entry;

        if (head == nullptr)
            tail = nullptr;
        return ret;
    });
}
```

Temporary entry pointer

Persistent Pop

```
int pop() {
    transaction::exec_tx(pool, [this] {
        auto                    e;
        auto                    ->next;

        delete_persistent<entry>(head);

        head = tmp_entry;

        if (head == nullptr)
            tail = nullptr;
        return ret;
    });
}
```

No object lifetime management

Persistent Pop

```
int pop() {
    transaction::exec_tx(pool, [this] {
        auto ret = head->value;
        auto tmp_entry = head->next;

        delete_persistent<entry>(head);

        head = tmp_entry;

        if (head == nullptr)
            tail = nullptr;
        return ret;
    });
}
```

Update head

Persistent Pop

Complete example

```
int pop(pool_base &pool) {
    transaction::exec_tx(pool, [this] {
        if (head == nullptr)
            throw runtime_error("Nothing to pop");

        auto ret = head->value;
        auto tmp_entry = head->next;
        delete_persistent<entry>(head);
        head = tmp_entry;
        if (head == nullptr)
            tail = nullptr;
        return ret;
    });
}
```

Transactional Allocation Functions

- 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

THREAD SYNCHRONIZATION

Persistent memory resident

Persistent Memory Synchronization

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

ALLOCATOR

Details

- Standard compliant implemetation
  - allocate/deallocate
  - construct/destroy
  - max_size
  - rebind
- Uses persistent_ptr as its pointer type (really important)
- Transactional only
- Not much left to say...

CONTAINERS ENABLING

Why?

- The pointer type is implemented
- The allocator is implemented
- The std containers are already there
  - Almost immediately usable
- Widely used with a familiar interface
- Maintained outside of NVML

std::vector

```
typedef typename __alloc_traits::pointer      pointer;

pointer                                       __begin_;
pointer                                       __end_;
pointer                                       __end_cap_;
```

std::map (AKA RB Tree)

```
template <class _VoidPtr>
class __tree_node_base
{
    // many other things
    pointer           __right_;
    __parent_pointer __parent_;
    bool __is_black_;
```

std::map (AKA RB Tree)

```
template <class _VoidPtr>
class __tree_node_base
{
    // many other things
    pointer           __right_;
    __parent_pointer __parent_;
    bool __is_black_;
```

Vital node metadata

std::map (AKA RB Tree)

```
template <class _VoidPtr>
class __tree_node_base
{
    // many other things
    pointer           __right_;
    __parent_pointer __parent_;
    bool __is_black_;
```

Will not get snapshot

std::map (AKA RB Tree)

```
template <class _VoidPtr>
class __tree_node_base
{
    // many other things
    pointer           __right_;
    __parent_pointer __parent_;
    bool __is_black_;
```

Where's the allocator?

<memory>

```
template <class _Ptr>
struct pointer_traits
{
    typedef _Ptr                pointer;

    // many other things

    typedef typename __pointer_traits_persistency_type<element_type,
        pointer>::type          persistency_type;
```

Injects p<>

std::map revisited

```
template <class _VoidPtr>
class __tree_node_base
{

typedef typename __rebind_persistency_type<pointer, bool>::type bool_type;
// many other things
bool_type __is_black_;
```

std::map revisited

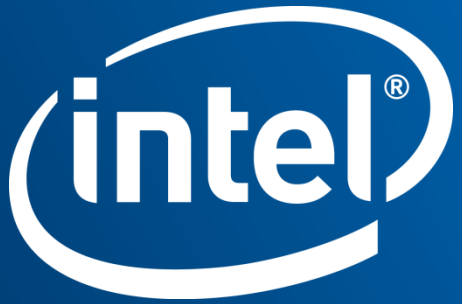
```
template <class _VoidPtr>
class __tree_node_base
{

typedef typename __rebind_persistency_type<pointer, bool>::type bool_type;
// many other things
bool_type __is_black_;
```

p<bool>

KNOWN ISSUES

Issues

- Static data members
  - The DATA/BSS section is not in persistent memory
- Standard library layout versioning
- Reliable vtable data rebuild
- No std::string equivalent - yet
- No persistent references – core language change


intel®

Handy links

Persistent memory programming homepage:

http://pmem.io

Intel developer zone:

https://software.intel.com/en-us/persistent-memory

Libpmemobj++ documentation:

http://pmem.io/nvml/cpp_obj/

Pmem Google Groups page:

https://groups.google.com/forum/#!forum/pmem