C++ Extensions For Persistent Memory

TOMASZ KAPELA

Intel
tomasz.kapela@intel.com

April 30, 2018

Abstract

Persistent memory technologies are becoming more popular. In addition to existing products like nonvolatile dual in-line memory module (NVDIMM-N) and dynamic random-access memory (DRAM) with NAND flash-based storage, there are new technologies emerging, like 3D XPoint™ memory. These types of hardware open interesting new possibilities along with many challenges. Persistent memory programming is fundamentally different from traditional programming to volatile memory due to the data retention after power failure. Intel® developed a set of libraries to help lessen the programming effort needed to switch to persistent memory. This paper describes the C++ API for one of those libraries and other proposed changes to the C++ standard.

I. INTRODUCTION

The Persistent Memory Development Kit (PMDK) consists of several separate libraries, each designed with a specific usage in mind. The most versatile in the suite is libpmemobj, which enables a safe and convenient way of leveraging the unique properties of persistent memory. However, libpmemobj is written in pure C without any language and compiler extensions. This makes the library easily accessible, since you do not need to use custom compilers; at the same time, it makes it hard to use and is error prone. We decided to alleviate these pain points by using higher-level languages that contain the necessary meta-programming features. The natural, first candidate was C++. It is highly versatile, rich in features, and is constantly improved with upcoming updates of the standard. The first step was to enable all of the functionality of the C implementation; the second was to let it interact with other C++ standard library components. This paper explains the approach taken and design decisions made during the implementation of the C++ version of libpmemobj.

II. TRANSACTIONS

Transactions are at the very core of all complex libpmemobj operations. The current x86-64 CPU architecture guarantees persistent atomicity only for 8-byte stores. What this means is that a single 8-byte store to memory cannot be torn. In case of power failure, you either get the old or new 8 bytes, never a mix. This is however inadequate for most algorithms used today. Being able to modify more than 8 bytes of storage at a time atomically is imperative for most non-trivial algorithms one might want to use in persistent memory. Commonly, a single logical operation requires multiple stores. For example, an insert into a simple list-based queue requires two separate stores: a tail pointer, and the next pointer of the last element. To enable the user to change larger portions of data atomically, with respect to power fail interruptions, the PMDK library introduces transactions. The C++ language bindings wrap these transactions into two concepts:
one, based on the Resource Acquisition Is Initialization (RAII) idiom, the other based on a callable std::function object. Additionally, because of some C++ standard issues, the scoped transactions come in two flavors: manual and automatic.

i. Manual Transactions

This is the simplest version of scope-based transactions available in the C++ implementation of libpmemobj. At the same time, they are the error prone to use. A normal code pattern looks like this.

```
Listing 1: Manual transaction
{
    pmem::obj::transaction::manual tx(pop);
    // do transactional work
    pmem::obj::transaction::commit();
}
```

Each transaction scope must end with a call to `pmem::obj::transaction::commit()`. It was our decision that by default the transaction would abort in the destructor. This is because the abort case is the unlikely case. If, by programming error in the error handling path, the developer forgot to explicitly call `pmem::obj::transaction::abort()` and the transaction would by default commit, that error would be hard to find. If the transaction aborts by default, the lack of `pmem::obj::transaction::commit()` is very likely to be found by basic smoke or unit tests. Executing any instruction within a transaction after the call to `pmem::obj::transaction::commit()` results in undefined behavior. This is true for all types of transactions. Returning from within an active transaction also results in undefined behavior. A return statement can only be executed within a closure-based transaction. The reason why this is not done automatically by the library is that it is impossible to infer whether you are leaving scope because of stack unwinding due to an exception or you are actually part of the cleanup routine. The C++ standard does provide `std::uncaught_exception()`, but it does not provide you with the number of active exceptions, which is necessary to be able to implement the automatic commit/abort logic in the destructor.

Following the golden rule to never throw from a destructor, there is no way to automatically inform the user about the status of the previous transaction. Therefore, there is a need to introduce an additional API to retrieve the status of the previously executed transaction. In libpmemobj, transactions are executed on a per-thread basis, hence the call returns the status of the last transaction performed by the calling thread. This is unfortunately not the ideal way of handling error cases, as it places an additional burden on the application developer.

```
Listing 2: Last transaction status
{
    pmem::obj::transaction::manual tx(pop);
    // do transactional work
    pmem::obj::transaction::commit();
}
```

auto error = pmem::obj::transaction::get_last_tx_error();

ii. Automatic Transactions

The C++ standard committee is aware of the inadequacy of `std::uncaught_exception()` and the issue is addressed in the new revision of the standard called C++17. In the improved version, `std::uncaught_exceptions()` returns the number of active exceptions. The RAII transaction can now automatically decide, upon scope exit, whether to abort or commit the transaction. This can be done by reading the number of active exceptions in the constructor of the transaction object and again in the destructor. If the latter is greater than the former, it means that the transaction object is being destroyed due to exceptional stack unwinding and the transaction needs to be aborted; otherwise, it can be safely committed. This removes the burden from the developer to always remember to commit the transaction. This can be done by reading the number of active exceptions in the constructor of the transaction object and again in the destructor. If the latter is greater than the former, it means that the transaction object is being destroyed due to exceptional stack unwinding and the transaction needs to be aborted; otherwise, it can be safely committed. This removes the burden from the developer to always remember to commit the transaction. The resulting code is cleaner and easier to maintain. The automatic transaction throws an exception from the destructor if it is aborted and the user is no longer burdened with having to explicitly call `get_last_tx_error()`.
iii. Function Object Transactions

The most interesting type of transaction available is the one calling a function object. The method is declared as:

```cpp
void pmem::obj::transaction::exec_tx(pool_base &pop, std::function<void ()> tx, Locks&... locks);
```

Thanks to the `std::function`, a myriad of types can be passed into `exec_tx`. One of the preferred ways is to pass a lambda function as the `tx` parameter. This makes the code compact and easier to analyze.

Listing 4: Function object transaction

```cpp
// start a transaction
transaction::exec_tx(pop, [&] () { // do transactional work
});
```

Of course, this API is not limited to just lambda functions; any callable target can be passed to `exec_tx` - functions, bind expressions, and function objects, as well as pointers to member functions. Since `exec_tx` is a normal static member function, it has the benefit of being able to throw exceptions. If an exception is thrown during the execution of a transaction, it is automatically aborted and the active exception is rethrown so information about the interruption is not lost. If the underlying C library fails for any reason, the transaction is also aborted and a C++ library exception is thrown. The developer is no longer burdened with the task of checking the status of the previous transaction. This type of transaction is the most versatile of the ones presented in this document; although, depending on the coding style, they may lead to excessive code indentation.

iv. Transaction Synchronization

Modern systems systems are often multi-threaded. To ensure that data is consistent when modifying the same memory region from multiple threads, some mutual exclusion algorithm is often employed. Providing proper thread isolation in the presence of transactions that can arbitrarily be rolled back is complicated. The locks need to be held until the end of the transaction, in case the modified objects need to be reverted to their original state. All of the aforementioned transaction types provide an entry point for persistent memory-resident synchronization primitives like `pmem::obj::mutex`, `pmem::obj::shared_mutex`, and `pmem::obj::timed_mutex`. Libpmemobj ensures that all locks are properly reinitialized when one attempts to acquire a lock for the first time. The usage of pmem locks is completely optional and transactions can be executed without them. The number of supplied locks is arbitrary and the types can be freely mixed.

Listing 5: Adding transactional synchronization

```cpp
// start transactions with synchronization
transaction::exec_tx(pop, worker_fun,
    a_mutex, a_shared_lock, a_timed_mutex);
transaction::manual(pop,
    a_mutex, a_shared_lock, a_timed_mutex);
transaction::automatic(pop,
    a_mutex, a_shared_lock, a_timed_mutex);
```

The locks are held until the end of the given transaction, or the outermost transaction in the case of nesting. That means that when transactions are enclosed by a try-catch statement, the locks are released before reaching the catch clause. This is extremely important in case some kind of transaction abort cleanup needs to modify the shared state. In such a case, the necessary locks need to be reacquired in the correct order.

III. The p<> Template

The PMDK transactions are a mix of redo and undo logging. The redo log is used for modifying internal library metadata, while undo logs are used to snapshot user data. The C++ for PMEM
library requires manual snapshots before modifying data in a transaction. The C++ bindings do all of the snapshotting automatically, to reduce the probability of programmer error. The `pmem::obj::p` template wrapper class is the basic building block for this mechanism. It is designed to work with basic types and not compound types such as PODs or classes. This is because it does not define `operator->()` and there is no possibility to implement `operator()`. The implementation of `pmem::obj::p` is based on the `operator=()`. Each time the assignment operator is called, it means that the value wrapped by `p` will be changed and the library needs to snapshot the old value.

Listing 6: Examples of `p` usage

```c++
struct bad_example {
  int some_int;
  int some_float;
};
struct good_example {
  pmem::obj::p<int> pint;
  pmem::obj::p<float> pfloat;
} pmem::obj::p<bad_example> bad;
pmem::obj::p<good_example> good;
bad.pint = 2; // does not compile
good.pint = 5; // works
```

This does not mean that `pmem::obj::p` cannot be used to wrap whole structures. There are cases where such an approach is preferred where performance is a key factor; this is because it might prove to be faster to snapshot the whole object, rather than each of its members separately. Because the `pmem::obj::p` template class was designed to wrap basic type data, it should resemble the wrapped type as much as possible. This means that, for example, arithmetic types such as `pmem::obj::p<int>` should behave like an int. All of the necessary overloaded operators are defined in the `pext.hpp` header file.

Listing 7: Examples of `p` arithmetics

```c++
void foo(long param);
pmem::obj::p<int> pint = 5;
int val1 = 12;
pint = val1 + 3;
pint /= val1;
foo(pint);
std::cout << pint << std::endl;
```

One should avoid placing variables encapsulated by `pmem::obj::p` and `pmem::obj::persistent_ptr`, which will be described in a later section, on the stack, especially within transactions. This is because additional computations have to be made to determine the origin of the memory and whether or not to snapshot the data.

Listing 8: Examples of `p` stack usage

```c++
struct foo {
  pmem::obj::p<int> pint;
};
pmem::obj::exec_tx(pop, [] {
  foo stack_foo(5);
  stack_foo.pint += 42;
  // stack_foo used in a generic
  // bar algorithm
  bar(stack_foo);
});
```

The `pmem::obj::p` has two member functions for retrieving a reference and a const reference to the held value, defined as `T &get_rw();` and `const T &get_ro() const noexcept;`, Where `T` is the type of the held value. The `get_rw` method assumes the value will be changed and tries to snapshot it before returning the reference.

The `pmem::obj::p` does not add to the wrapped type’s size footprint. This is especially important for pools that are meant to be interoperable between the C and C++ API versions.

### IV. Persistent Pointer

The persistent memory programming model proposed by the Storage Networking Industry Association (SNIA), which is used by PMDK, is based on the notion of memory mapped files. Since files can be mapped at different addresses of the process address space, traditional pointers that store absolute addresses cannot be used without modification. It is possible that, after reopening the pool, said pointer might point to uninitialized memory and dereferencing it may result in a segmentation violation. And that is the optimistic case; in a pessimistic scenario it might point to a valid memory range, but not one that the user expects it to point to. Therefore, PMDK introduced a different type of pointer, called in the C library `struct pmemoid`. It consists of two 8-byte fields,
a pool identifier in the form of a shortened 16-byte UUID (Universally Unique Identifier), and an offset to the beginning of the pool. The `C++: pmem::obj::persistent_ptr` is a wrapper around the C struct `pmemoid`, and it has the exact same 16-byte footprint. It is in fact byte-compatible, and can be easily passed to the C API using the `const PMEMoid &raw()` member function. This property is especially important for pools that are meant to be opened both by C and C++ applications. The `pmem::obj::persistent_ptr` is similar in concept and API to the myriad of smart pointers introduced in C++11 (for example, `std::shared_ptr`), with one notable difference - it does not manage the object’s lifecycle. The object cannot be destroyed when the pointer goes out of scope, because that way after the program ends, all objects managed by the pointers would be deallocated. After reopening the memory pool there would be no state to reload, hence rendering the most interesting property of this new type of memory, its persistence, useless. The `pmem::obj::persistent_ptr` has similar properties and behavior to traditional pointers and the `std::shared_ptr`. It is possible to point to compatible types; for example, if you have an inheritance hierarchy, where `foo` inherits from `bar`, a `pmem::obj::persistent_ptr<bar>` can point to a `foo` object. Similarly, in the case of multiple inheritance, the pointer will be offset to point to the appropriate base class. Both of these properties are shown in Listing 9.

**Listing 9: Persistent_ptr inheritance example**

```cpp
using namespace pmem::obj;

struct A {
    uint64_t a;
};
struct B {
    uint64_t b;
};
struct C : public A, public B {
    uint64_t c;
};

transaction::exec_ts(pop, &pop) {
    auto cpotr = make_persistent<C>();
    persistent_ptr<B> bpotr = cpotr;
    assert((bpotr.raw().off - cpotr.raw().off) == sizeof(A));
}
```

The pointer is convertible to and from `pmem::obj::persistent_ptr<oid>` for use in generic algorithms. It also provides the `reference_type operator*()` const and `pointer_type operator->()` const functions for dereferencing and member access, respectively. The `pmem::obj::persistent_ptr` also implements the subscript operator `reference_type operator[](std::ptrdiff_t i)` const noexcept for all types except `void`. The subscript operator also validates whether the given index is within the bounds of the pointed-to array. Normal pointer arithmetic is also allowed on the `pmem::obj::persistent_ptr`, as it is a `random_access_iterator`. It defines all typedefs and functions mandated both by `std::pointer_traits` and `std::iterator_traits`. This means that the `pmem::obj::persistent_ptr` type can be used in standard library containers and algorithms. Aside from all the standard typedefs, the persistent pointer also defines a new type - the `persistency_type using persistency_type = p<T>`. This typedef is necessary for standard library containers to properly manage their metadata in persistent memory. This will be explained later in this article in the section Changes to the C++ Standard Library.

### i. Allocating Functions

As with `std::shared_ptr`, the `pmem::obj::persistent_ptr` comes with a set of allocating and deallocating free functions. These help allocate memory and create objects, as well as destroy and deallocate the memory. This is especially important in the case of persistent memory, because all allocations and object construction/destruction must be done atomically with respect to power fail interruptions. These functions are subdivided into four header files:

- `make_persistent.hpp` - transactional allocation and deallocation of non-array types,
- `make_persistent_array.hpp` - transactional allocation and deallocation of array types,
- `make_persistent_atomic.hpp` - atomic allocation and deallocation of non-array types,
- `make_persistent_array_atomic.hpp` - atomic allocation and deallocation of array types.
i.1 Transactional API

The transactional allocations use perfect forwarding and variadic templates for object construction. This makes object creation similar to calling the constructor and identical to `std::make_shared`. The transactional array creation, however, requires the objects to be default constructible. The created arrays can be multidimensional, as shown in Listing 10. The `pmem::obj::make_persistent` and `pmem::obj::make_persistent_array` must be called within a transaction, otherwise an exception is thrown. During object construction, other transactional allocations can be made, and that is what makes this API very flexible. The specifics of persistent memory required the introduction of the `pmem::obj::delete_persistent` function, which destroys objects and arrays of objects. Since the `pmem::obj::persistent_ptr` does not automatically handle the lifetime of pointed-to objects, the user is responsible for disposing of the ones that are no longer in use. What is also new is the fact that transactional object destruction may fail because a deallocation has to be added to an expanding undo log. This is especially cumbersome when an object is responsible for deallocating other objects upon its destruction. This might cause an exception to be thrown from within a destructor. Since C++11 destructors are by default marked as `noexcept(true)`, the compiler is free to make optimizations based on that contract. Moreover, if the destructor is called due to stack unwinding because an exception was thrown, the `pmem::obj::delete_persistent` might throw another exception which, if uncaught, will result in a call to `std::terminate` and terminate the application.

Listing 10: Make_persistent example
```
// calls the foo(int, int) constructor
auto ptr1 = make_persistent<foo>(3, 4);
// calls the foo(double) constructor
auto ptr2 = make_persistent<foo>(3.14);
// array construction auto
arr1 = make_persistent<foo[]>(6);
auto arr2 = make_persistent<foo[]>(1);
auto arr3 = make_persistent<foo>[3][2][3](5);
// deletes objects
delete_persistent<foo>(ptr1);
delete_persistent<foo>(ptr2);
```

i.2 Atomic API

The atomic allocations behave differently in a number of aspects. First of all, they do not return a pointer; the user has to provide a reference to one as the function's argument. Because atomic allocations are not executed in the context of a transaction, the actual pointer assignment must be done through other means; for example, by redo logging the operation. The `pmem::obj::make_persistent_atomic` function also accepts an arbitrary number of constructor arguments; however, the method of passing those arguments is different. They have to be passed into the C API as `void *arg`. This is done by packing all the arguments into a `std::tuple` and unpacking them in the C-style constructor function, which in turn passes the arguments into the actual object's constructor. Despite a considerable difference in implementation, both the transactional and atomic allocations behave similarly. The atomic array allocations share the same constraints as their transactional counterparts. There are two major differences in how both types of allocations behave. First, within the atomic allocation's constructor, no additional object construction is allowed. The second difference between the atomic and transactional API is in the way `pmem::obj::delete_persistent` and `pmem::obj::delete_persistent_atomic` work. The atomic version does not call the object's destructor. This is because the whole operation has to be atomic, and there is no way to back up the destroyed object's data other than by transactional snapshotting. Atomic allocations should not be used within a transaction. To sum up, the transactional allocation/deallocation is more flexible, enables allocation nesting, and calls destructors. All of this makes the transactional API slightly slower. When speed is of the utmost importance, it might prove beneficial to implement the time-constrained algorithm using the atomic API.
### ii. Polymorphism

One major drawback of the current implementation of the C++ bindings to the PMDK libpmemobj is that it does not allow the `pmem::obj::persistent_ptr` to point to polymorphic types. This is because we know of no reliable and portable way to implement vtable rebuilding after reopening the pool. This rule is enforced by the implementation of `pmem::obj::persistent_ptr`, so that the user does not destroy a pool by accident. If some day we get a standard extension that makes vtable rebuilding a reality, this feature, when implemented by us, might break interoperability between C and C++ programs that leverage it.

### V. Allocator

The allocator concept has been part of the C++ standard since the original STL implementation done by Alexander Stepanov and Meng Lee. The allocator concept enables the developer to implement a substitute for the memory allocation model of the standard library’s containers. Due to this feature, each container can carve out fragments of a persistent memory pool in order to store user data. This powerful concept, together with transactions and the persistent pointer, enables the standard library’s containers to reside and safely operate on a persistent memory medium. The persistent memory allocator implements the mandatory interface defined by the `std::allocator_traits` in the C++ standard section [allocator.traits]. It can be used both to allocate/deallocate memory as well for object construction and destruction. All of these operations have to be performed within an active transaction, otherwise an appropriate exception is thrown. As such all of the memory manipulation done by this allocator is power-fail safe. Due to this consistency constraint, the performance of the allocator suffers a slight penalty. However, due to the specifics of persistent memory, this compromise is necessary. If all operations were not enclosed within transaction, there is a possibility of memory corruption and permanent leaks.

For the initial implementation, because this is a highly specialized allocator designed to be used in a particular context: `propagate_on_container_copy_assignment`, `propagate_on_container_move_assignment`, and `propagate_on_container_swap` are set to `false_type`, hence disallowing allocator copy/move/swap between containers. This will cause undefined behavior if two containers with unequal allocators are swapped. The allocator itself has a modular design, allowing the user to modify specific parts of the allocation/construction process. The Policy template parameter is responsible for memory allocation and deallocation, and implements the `pointer allocate(size_type cnt, const void* hint = 0)` function to allow memory to be allocated for a given object of type T. The default implementation does not use the hint argument, as there is no allocation locality support implemented in libpmemobj. The allocator is a crucial component in implementing persistent memory support within the C++ standard library, especially for containers described in the [containers] section of the standard. In C++17, there is the notion of `std::pmr::polymorphic_allocator` and `std::pmr::memory_resource`.  

---

**Listing 11: Make_persistent example**

```cpp
persistent_ptr<foo> ptr1;
persistent_ptr<foo> ptr2;
persistent_ptr<foo[]> arr1;
persistent_ptr<foo[5]> arr2;
persistent_ptr<foo[1][2][3]> arr3;

make_persistent_atomic<foo>(pop, ptr1, 2, 3);
make_persistent_atomic<foo>(pop, ptr1, 2.2);

make_persistent_atomic<foo[]>(pop, arr1, 6);
make_persistent_atomic<foo[5]>(pop, arr2);
make_persistent_atomic<foo[1][2][3]>(pop, arr3, 5);

// deletes objects
delete_persistent_atomic<foo>(ptr1);
delete_persistent_atomic<foo>(ptr2);
delete_persistent_atomic<foo[]>(arr1, 6);
delete_persistent_atomic<foo[5]>(arr2);
delete_persistent_atomic<foo[1][2][3]>(arr3, 5);
```
C++ for PMEM

It’s an allocator that exhibits different behavior depending on the type of std::pmr::memory_resource it was constructed with. It is however not suitable to be used in persistent memory, because std::pmr::memory_resource is an abstract interface used to define a concrete memory resource. This would cause a vtable to be generated and, as it was previously mentioned, polymorphic types are not supported at this moment.

VI. CHANGES TO THE C++ STANDARD LIBRARY

i. Containers

One of the most important parts of the C++ standard library is the containers library described in the [containers] section. Containers manage the lifetime of held objects through allocation/creation, deallocation/destruction with the use of allocators. For prototyping support for persistent memory in the standard library, the LLVM libcxx release 3.9 was chosen. The choice was made after an initial prototype was made, showing that libcxx is easier to adapt, given its younger code base. There are a number of changes made to the standard library, not all strictly related to persistent memory. The std::deque container had strong const requirements, which had to be relaxed to be able to use fancy pointers as the container’s allocator’s pointer type.

Listing 12: Deque changes

```cpp
typedef typename __rebind_alloc__<
  __alloc_traits, pointer>::type
__pointer_allocator; typedef
__pointer_allocator
__map_traits; typedef typename __map_traits
  ::pointer
__map_pointer;

- typedef typename __rebind_alloc__<
  __alloc_traits, const_pointer>::type
__const_pointer_allocator; - typedef
  typename allocator_traits<
  __const_pointer_allocator>::
  __const_pointer __map __const_pointer;

- typedef typename allocator_traits<
  __pointer_allocator>::split_buffer
  <pointer,
  __pointer_allocator> __map;
```

Some containers, like std::vector, work well without any changes in the standard library. This is because the implementation is based on the std::allocator_traits::pointer type. All of the allocations are done by the substituted allocator through these pointers. Therefore, if all std::vector operations are enclosed within a transaction, it can be used in a power-fail safe way.

Not all containers are so simple, and most of them require some internal metadata to function properly. For example, std::map, which is implemented as a red-black tree, requires each node to remember its color. In libcxx this is represented by a bool __is_black field in the class __tree_node_base. During tree rebalancing this field can change and thus needs to be tracked by a transaction. For this purpose, the pmem::obj ::p wrapper class needs to be used.

The standard library containers have only one common entry point for supplying a custom allocator. The allocator would be a good place to inject the new wrapper type. Some implementations, however, are oblivious to the allocator and have no way to extract the knowledge of the type of memory they are allocated from. This is the case with the aforementioned class __tree_node_base, which only has knowledge of the void pointer type. Therefore, even though the allocator would have been a better place to supply the information about the persistent memory property, a decision was made to implement it in the pointer type associated with the allocator.

The C++17 standard will introduce the std::pmr::memory_resource, which logically seems to be the best place to put such information; however, that would not immediately address the issue mentioned earlier. Additionally, as was previously noted, the std::pmr::memory_resource is an abstract interface, and due to the vtable reconstruction problem is not a viable solution at the moment. One of the main assumptions of this prototype is that it has to be compatible with all the existing code to date. Especially, the typical usage of standard library containers with the std::allocator cannot be affected by the changes made to the standard library. There-
C++ for PMEM

fore, if the std::allocator_traits::pointer does not define its own concept of persistency, a default, backward-compatible type has to be provided automatically. This is a concept that is commonly used throughout the standard library and its implementation can be seen in Listing 13. Naturally, the new typedef has to be introduced into the std::pointer_traits, as well as to the native pointer type specialization pointer_traits<_Tp*> for the traits.

Listing 13: __has_persistency_type implementation

```cpp
template <class _Ptr>
struct __has_persistency_type {
private:
    static _two { char _lx; char _lx;};
    template <class _Up> static _two __test (...);
    template <class _Up> static char __test (typename _Up::persistency_type* = 0);
public:
    static const bool value =
    sizeof(__test<_Ptr>(0)) == 1;
};
template <class _Tp, class _Ptr, bool = __has_persistency_type<_Ptr>::value>
struct __pointer_traits_persistency_type {
    typedef _Tp type;
};
template <class _Tp, class _Ptr>
struct __pointer_traits_persistency_type<_Tp, _Ptr, true> {
    typedef typename _Ptr::persistency_type type;
};
```

This is the foundation for supplying the notion of persistency to the standard library containers. The new typedef is called persistency_type and for the pmem::obj::persistent_ptr it is defined as using persistency_type = p<T>. For the standard pointer type it would be using persistency_type = T. The mechanism for changing the type for which the persistency concept is to be applied is based on the rebindability of the pointer:

```cpp
using pmem = std::map<int, foo, std::less<int>>;

// allocate a new persistent map object
auto pmmap = make_persistent<pmap>();
// use the persistent map std::map
pmmap->insert(std::make_pair(1, foo()));
// pmmap is a pointer
auto pmmap = pmmap->insert(make_pair(1, foo(11)));
```

Applying all of the above-mentioned concepts to the implementation of class __tree_node_base yields the changes shown in Listing 14. The __rebind_persistency_type is a helper template that rebinds the pointer to type bool and returns the std::pointer_traits<pmem::obj::persistent_ptr<bool>>::persistency_type, which boils down to pmem::obj::p<bool>.

Listing 15: std::map usage

```cpp
using pmap = std::map<int, foo, std::less<int>>;

// open a preexisting pool with layout "map Example"
auto pop = pool<someroot>::open(path, "map_example");

// start a transaction
transaction::exec tx(pop, [&pop] {
    // allocate a new persistent map object
    auto pmmap = make_persistent<pmap>();
    // use the persistent map std::map
    pmmap->insert(std::make_pair(1, foo()));
    // pmmap is a pointer
    (*pmmap)[2] = foo();
    pmmap->erase(2);
    pmmap->insert(pmmap->begin(), std::make_pair(1, foo(11)));
    pmmap->begin()->second = foo(234);
});
```

Thanks to these changes, the std::map is now able to correctly work with the pmem::obj::allocator in a persistent memory-specific environment. An example of std::map usage can be seen in Listing 15. All of the changes made to libcxx can be found on the libcxx fork on GitHub under the PMEM organization.

ii. Object lifetime

The C++ standard in section [basic.life] describes the lifetime as a runtime property of an object. The standard states that the lifetime of an object starts when storage with proper alignment is obtained, and if the object has non-vacuous initialization, its initialization is complete. Citing the standard, an object is defined to have non-vacuous initialization if it is of a class or aggregate type, and it or one of its members is initialized by a constructor other than a
trivial default constructor. Later on, the standard states that properties ascribed to objects throughout it apply for a given object only during its lifetime. The Storage Networking Industry Association non-volatile memory (NVM) Programming Model uses memory mapped files, the contents of which might have been populated with objects in a prior execution of a program. This is a similar problem to transmitting data over a network, where the C++ application is given an array of bytes, but might be able to recognize the type of object sent. However, the object was not constructed in this application, so using it would result in undefined behavior.

This is a known problem that is being addressed by WG21. This does not solve all of the problems because, ideally, we would like complex and polymorphic objects to reside in persistent memory.

**VII. Other Challenges and Conclusion**

For this programming concept to be useful to the general public, some kind of formal description or standardization of the aforementioned concepts and features should take place. There are however many issues that were not addressed by this paper but are otherwise not implemented or handled by the underlying C library; these are feature compatibility, architecture compatibility, layout, and library versioning, to name a few. One of the main issues that never before had to be addressed is standard library data layout versioning. Since persistent memory retains its contents, the on-media layout version of in-memory structures has to be provided to the library’s user for persistent memory pool verification upon subsequent opens. Otherwise, should the application be recompiled with a newer version of the library with a modified layout, the loaded data would be different and the whole pool could be corrupted. What is more, the format should not change with different compiler flags, or the compiler flags should be easily readable by the resulting binary. This can happen, for example, when the -fshort-enums flag is provided to the GNU Compiler Collection (GCC).

Persistent memory programming is a fairly new concept and is very different from traditional software development. It comes with a set of its own advantages and challenges. It does however promise a performance improvement in some workloads and use cases, such as instantaneous database loading. It comes with a few drawbacks, like added runtime complexity, because of the need for providing transactional consistency.

**VIII. Notices**

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document. Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Intel, the Intel logo, and 3D XPoint are trademarks of Intel Corporation in the U.S. and/or other countries.

This sample source code is released under the Intel Sample Source Code License Agreement.

©2018 Intel Corporation