PERSISTENT MEMORY EXTENSIONS TO LIBSTDC++/LIBC++

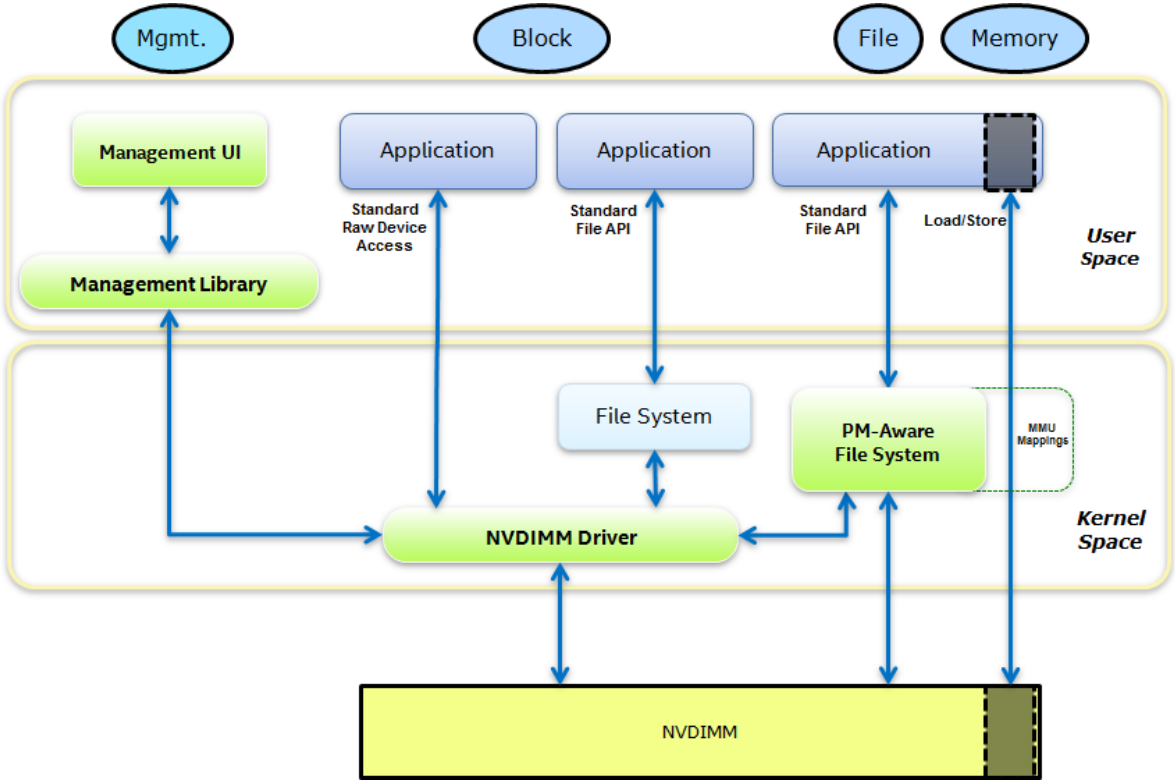
Tomasz Kapela, @tomaszkapela

LinuxCon Europe, Berlin 2016

THE MODEL

Hype

The NVM programming model



NVML

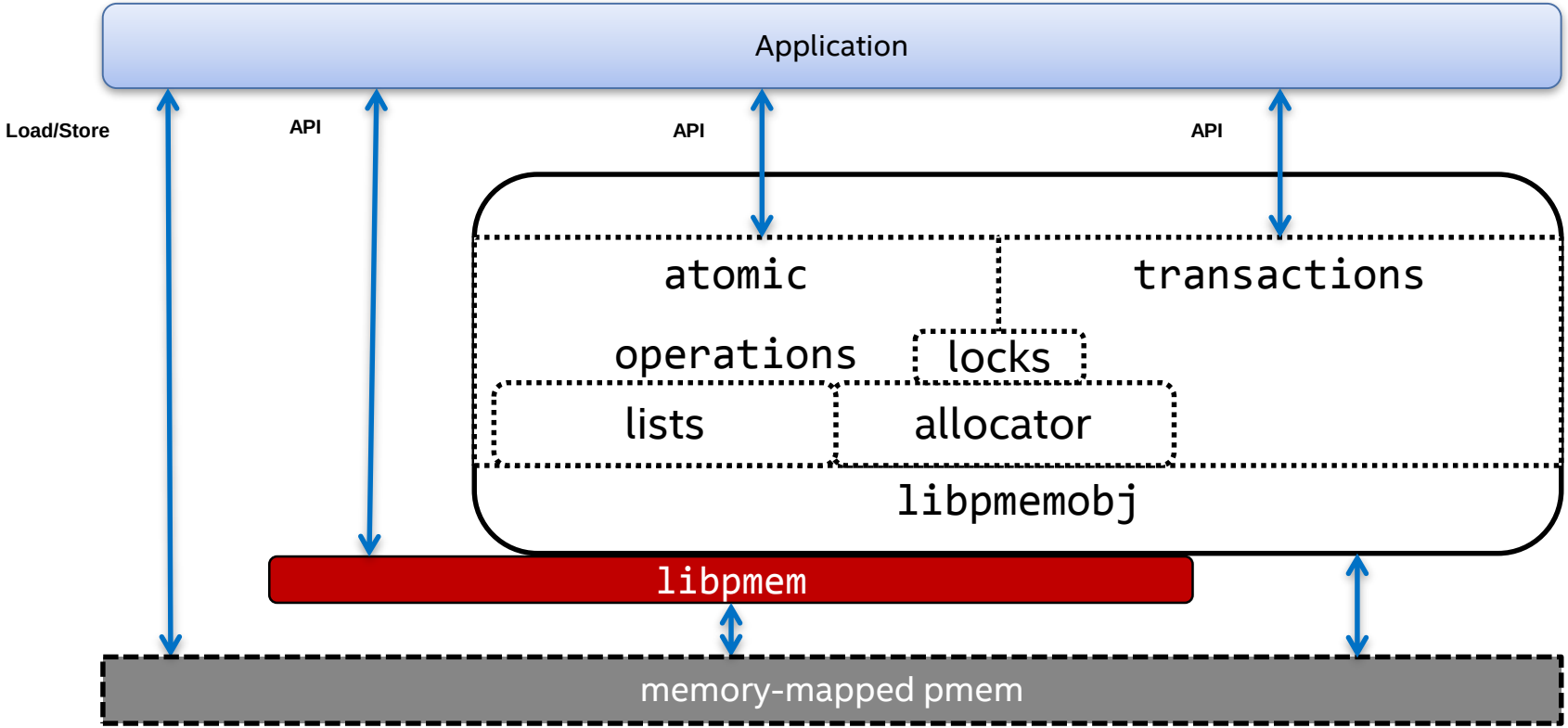
The solution

NVML

Open source set of libraries available at www.github.com/pmem/nvml

Library	Description
libpmem	The basic library, flush to persistence
libvmem	Volatile Memory Allocator
libvmmalloc	Transparent use of libvmem
libpmemblk	Persistent memory carved into blocks
libpmemlog	Log file (append-mostly)
libpmemobj	Transactional Object Store

libpmemobj


Application
Load/Store
API
API
API
atomic
transactions
operations
locks
lists
allocator
libpmemobj
libpmem
memory-mapped pmem

PAIN POINTS

Pointers by libpmemobj

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
/*
 * Object handle
 */
typedef struct pmemoid {
        uint64_t pool_uuid_lo;
        uint64_t off;
} PMEMoid;
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
/*
 * Object handle
 */
typedef struct pmemoid {
        uint64_t pool_uuid_lo;
        uint64_t off;
} PMEMoid;
```

```
void *gen_ptr;
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
/*
 * Object handle
 */
typedef struct pmemoid {
        uint64_t pool_uuid_lo;
        uint64_t off;
} PMEMoid;
```

```
void *gen_ptr;
```

```
struct foo *type_ptr;
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
/*
 * Object handle
 */
typedef struct pmemoid {
        uint64_t pool_uuid_lo;
        uint64_t off;
} PMEMoid;
```

```
void *gen_ptr;
```

```
TOID(struct foo) persistent_ptr;
```

```
struct foo *type_ptr;
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
1  #include <libpmemobj.h>
2
3  /*
4   * Layout definition
5   */
6  POBJ_LAYOUT_BEGIN(pi);
7  POBJ_LAYOUT_ROOT(pi, struct pi);
8  POBJ_LAYOUT_TOID(pi, struct pi_task);
9  POBJ_LAYOUT_END(pi);
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary
- A lot of macros to make this feasible

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
89  #ifdef __cplusplus
90  #define _TOID_CONSTR(t)\
91  _toid_##t##_toid()\
92  { }\
93  _toid_##t##_toid(PMEMoid _oid) : oid(_oid)\
94  { }
95  #else
96  #define _TOID_CONSTR(t)
97  #endif
98
99  /*
100  * Declaration of typed OID
101  */
102 #define _TOID_DECLARE(t, i)\
103 typedef uint8_t _toid_##t##_toid_type_num[(i) + 1];\
104 TOID(t)\
105 {\
106         _TOID_CONSTR(t)\
107         PMEMoid oid;\
108         t *_type;\
109         _toid_##t##_toid_type_num *_type_num;\
110 }
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary
- A lot of macros to make this feasible
- A lot of macros to handle the TOID

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
106 /*
107  * btree_find -- searches for key in the tree
108  */
109 const char *
110 btree_find(PMEMobjpool *pop, int64_t key)
111 {
112         TOID(struct btree) btree = POBJ_ROOT(pop, struct btree);
113         TOID(struct btree_node) node = D_RO(btree)->root;
114
115         while (!TOID_IS_NULL(node)) {
116                 if (D_RO(node)->key == key)
117                         return D_RO(node)->value;
118                 else
119                         node = D_RO(node)->slots[key > D_RO(node)->key];
120         }
121
122         return NULL;
123 }
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary
- A lot of macros to make this feasible
- A lot of macros to handle the TOID
  - Why?

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

```
106 /*
107  * btree_find -- searches for key in the tree
108  */
109 const char *
110 btree_find(PMEMobjpool *pop, int64_t key)
111 {
112         PMEMoid btree = pmemobj_root(pop, sizeof(struct btree));
113         PMEMoid node = ((struct btree *)pmemobj_direct(btree))->root;
114
115         while (!OID_IS_NULL(node)) {
116                 if (((struct btree_node *)pmemobj_direct(node))->key == key)
117                         return ((struct btree_node *)pmemobj_direct(node))->valu
118                 else
119                         node = ((struct btree_node *)pmemobj_direct(node))->slot
120         }
121
122         return NULL;
123 }
```

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary
- A lot of macros to make this feasible
- A lot of macros to handle the TOID
  - Why?
- Is it usable?

The fancy typed pointer – TOID

Type safety introduced to a fancy based pointer.

- Type declarations necessary
- A lot of macros to make this feasible
- A lot of macros to handle the TOID
  - Why?
- Is it usable?

```
266     uint8_t *dst = D_RW(D_RW(bp)->data) + block_off;
```

THE C++ APPROACH

Addressing the pain points

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

- No explicit type declaration

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

- No explicit type declaration
  - typeid(T).hash_code()

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

- No explicit type declaration
  - typeid(T).hash_code()
- No more macros!

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

```
125         // No type declarations necessary
126         auto ptr = make_persistent<foo>(1, 'a'); // arguments forwarding
127
128         ptr->bar = 0xDEADBEEF; // normal member access
129         ptr->check_foo(2, 'b');
130
131         persistent_ptr<foo> conv = make_persistent<bar>(4); // foo <- bar
132
133         auto array = make_persistent<foo[10]>();
134         array[10].bar = 0xDEADBEEF; // runtime assert - out of bounds
135         array[2].check_foo(2, 'b');
136
137         delete_persistent<foo>(ptr);
138         delete_persistent<foo[10]>(array);
139         delete_persistent<foo>(conv);
```

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

- No explicit type declaration
  - typeid(T).hash_code()
- No more macros!
- Mimics a well known construct
  - std::shared_ptr
- Now is it usable?

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

```
106 /*
107  * btree_find -- searches for key in the tree
108  */
109 const char *
110 btree_find(PMEMobjpool *pop, int64_t key)
111 {
112         TOID(struct btree) btree = POBJ_ROOT(pop, struct btree);
113         TOID(struct btree_node) node = D_RO(btree)->root;
114
115         while (!TOID_IS_NULL(node)) {
116                 if (D_RO(node)->key == key)
117                         return D_RO(node)->value;
118                 else
119                         node = D_RO(node)->slots[key > D_RO(node)->key];
120         }
121
122         return NULL;
123 }
```

The fancy typed pointer – persistent_ptr

Encapsulate type info into a smart pointer.

```
110 /*
111  * btree_find -- searches for key in the tree
112  */
113 const char *
114 btree_find(pool<btree> &pop, int64_t key)
115 {
116         auto btree = pop.get_root();
117         auto node = btree->root;
118
119         while (node != nullptr) {
120                 if (node->key == key)
121                         return node->value;
122                 else
123                         node = node->slots[key > node->key];
124         }
125
126         return nullptr;
127 }
```

The fancy typed pointer – persistent_ptr

Known issues

- No polymorphic types allowed in persistent memory
  - Vtables rebuild issue
- typeid(T).hash_code()
  - Returns an unspecified value, except that within a single execution of the program, it shall return the same value for any two type_info objects which compare equal.

PAIN POINTS

Transactions by libpmemobj

Transactions

Consistency critical roll-back transactions.

- ACID like properties

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions

```
143         /* set the return point */
144         jmp_buf env;
145         if (setjmp(env)) {
146                 /* end the transaction */
147                 pmemobj_tx_end();
148                 return 1;
149         }
150
151         /* begin a transaction, also acquiring the write lock for the log */
152         pmemobj_tx_begin(pop, env, TX_LOCK_RWLOCK, &bp->rwlock, TX_LOCK_NONE);
```

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions

```
143         /* set the return point */
144         jmp_buf env;
145         if (setjmp(env)) {
146                 /* end the transaction */
147                 pmemobj_tx_end();
148                 return 1;
149         }
150
151         /* begin a transaction, also acquiring the write lock for the log */
152         pmemobj_tx_begin(pop, env, TX_LOCK_RWLOCK, &bp->rwlock, TX_LOCK_NONE);
```

```
177         pmemobj_tx_commit();
178         pmemobj_tx_end();
```

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions
- Macros to make this usable

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions
- Macros to make this usable

```
140         /* begin a transaction, also acquiring the write lock for the log */
141         TX_BEGIN_LOCK(pop, TX_LOCK_RWLOCK, &D_RW(bp)->rwlock, TX_LOCK_NONE) {
142
```

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions
- Macros to make this usable

```
140         /* begin a transaction, also acquiring the write lock for the log */
141         TX_BEGIN_LOCK(pop, TX_LOCK_RWLOCK, &D_RW(bp)->rwlock, TX_LOCK_NONE) {
142
```

```
164         } TX_ONABORT {
165                 retval = -1;
166         } TX_END
```

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions
- Macros to make this usable
- A bit error prone
  - Even with the macros

Transactions

Consistency critical roll-back transactions.

```
62          /* initialize pointer variables with invalid addresses */
63          int *bad_example_1 = (int *)0xBAADF00D;
64          int *volatile good_example = (int *)0xBAADF00D;
65
66          TX_BEGIN(pop) {
67                  bad_example_1 = malloc(sizeof(int));
68                  good_example = malloc(sizeof(int));
69
70                  /* manual or library abort called here */
71                  pmemobj_tx_abort(EINVAL);
72          } TX_ONABORT {
73                  /*
74                   * This section is not longjmp-safe
75                   */
76                  free(good_example); /* OK */
77                  free(bad_example_1); /* undefined behavior */
78          } TX_END
```

Transactions

Consistency critical roll-back transactions.

- ACID like properties
- „C” exceptions
- Macros to make this usable
- A bit error prone
  - Even with the macros
- Snapshotting

Transactions

Consistency critical roll-back transactions.

```
151     /* add the modified root object to the undo log */
152     TX_ADD(bp);
153     if (TOID_IS_NULL(D_RO(bp)->tail)) {
154             /* update head */
155             D_RW(bp)->head = logp;
156     } else {
157             /* add the modified tail entry to the undo log */
158             TX_ADD(D_RW(bp)->tail);
159             D_RW(D_RW(bp)->tail)->hdr.next = logp;
160     }
```

THE C++ APPROACH

Addressing the pain points

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction

```
345         {
346                 transaction::manual tx(pop);
347                 auto pfoo = make_persistent<foo>();
348
349                 pfoo->bar = 2; // no TX_ADD needed
350
351                 transaction::commit(); // explicit commit necessary
352         }
353         auto result = transaction::get_last_tx_error();
```

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction

```
355         transaction::manual tx(pop);
356         auto pfoo = make_persistent<foo>();
357         {
358                 transaction::manual tx(pop);
359
360                 pfoo->bar = 2;
361
362                 transaction::abort(EINVAL); // explicit abort
363         }
364
365         transaction::commit(); // this will still abort the outer transaction
```

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - CppCon2015/Declarative Control Flow - Andrei Alexandrescu

```
367         {
368                 transaction::automatic tx(pop); // available in C++17
369
370                 ++(pfoo->bar); // do work
371                 // no more explicit commit
372         }
```

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - Still can't throw an exception though

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - Still can't throw an exception though
- The lambda version

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - Still can't throw an exception though
- The lambda version

```
375         auto pfoo = make_persistent<foo>(); // runtime exception
376
377         transaction::exec_tx(pop, [&] {
378                 pfoo->bar += 12;
379                 delete_persistent<foo>(pfoo); // will roll back on abort
380         }, pfoo->smtx); // take arbitrary locks on tx begin
```

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - Still can't throw an exception though
- The lambda version

```
383         transaction::exec_tx(pop, [] {
384                 transaction::exec_tx(pop, [] {
385                         throw std::bad_alloc();
386                 }); // abort transaction and waterfall std::bad_alloc
387         }); // roll-back changes, rethrow std::bad alloc
```

Transactions

Making life easier with RAII and lambdas.

- Manual and automatic transaction
  - The problem with std::uncaught_exception
  - Still can't throw an exception though
- The lambda version
  - Self contained
  - Somewhat ugly looking

THE ALLOCATOR

Using the containers

Persistent memory containers

A proof of concept under way.

- Targeting libc++ and libstdc++

Persistent memory containers

A proof of concept under way.

- Targeting libc++ and libstdc++

```
329         typedef std::vector<foo, nvml::obj::allocator<foo>> pvector;
330
331         struct root {
332                 persistent_ptr<pvector> my_vector;
333         };
334
335         nvobj::pool<root> pop = nvobj::pool<root>::open(path, "layout");
336
337         transaction::exec_tx(pop, [&] {
338                 auto root = pop.get_root();
339
340                 root->my_vector->emplace_back(0xDEADBEEF);
341                 root->my_vector->push_back(foo(0xBADA55));
342
343                 for(auto el : root->my_vector)
344                         std::cout << el << std::endl;
345         });
```

Persistent memory containers

A proof of concept under way.

- Targeting libc++ and libstdc++
  - Libc++ is mostly working
  - Libstdc++ needs a bit more work

Persistent memory containers

A proof of concept under way.

- Targeting libc++ and libstdc++
  - Libc++ is mostly working
  - Libstdc++ needs a bit more work

```
/home/tkapela/install/gcc/include/c++/6.1.0/bits/stl_list.h:574:11: error: cannot convert 'nvml::obj::persistent_ptr<std::_List_node<{anonymous}::foo> >' to
_Node* {aka std::_List_node<{anonymous}::foo>*}' in return
    return __p;
           ~~
```

Persistent memory containers

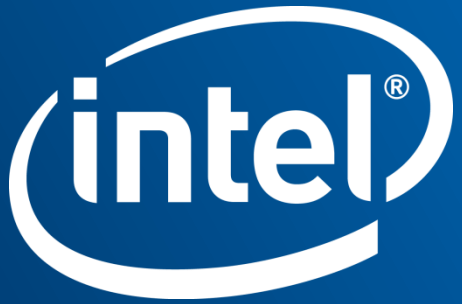
A proof of concept under way.

- Targeting libc++ and libstdc++
  - Libc++ is mostly working
  - Libstdc++ needs a bit more work
- Main pros
  - It's for free! (well, almost)
  - Familiar interface
  - Well tested implementation of containers

Persistent memory containers

What are the known issues

- Internally held data is not „aware” of persistence
- T* used instead of allocator_traits::pointer
- Layout versioning
- Current implementation works only on x86_64


intel®

Handy links

The allocator branch:

https://github.com/tomaszkapela/nvml/tree/cpp-allocator

Libpmemobj++ documentation:

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

Pmem Google Groups page:

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