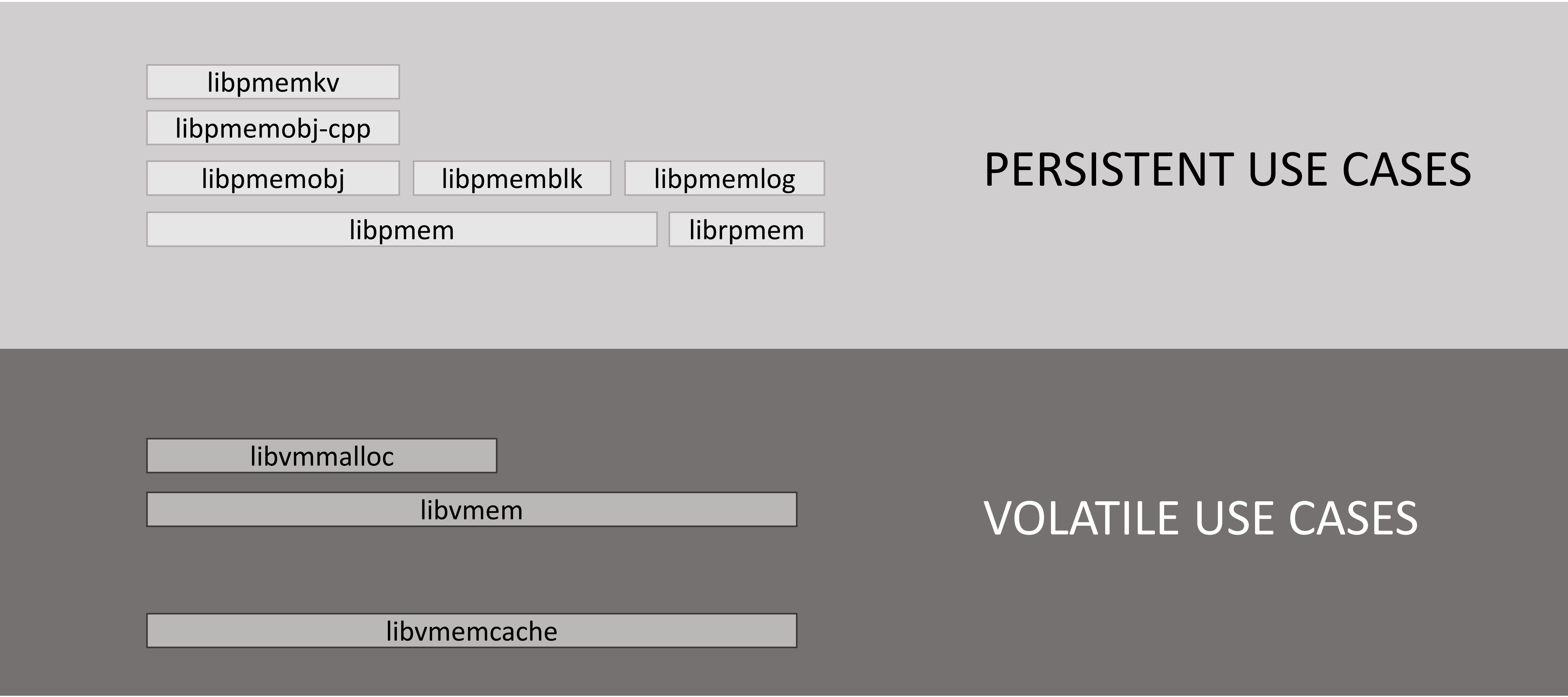
Evolution of PMDK

July 2019

Piotr Balcer
<piotr.balcer@intel.com>

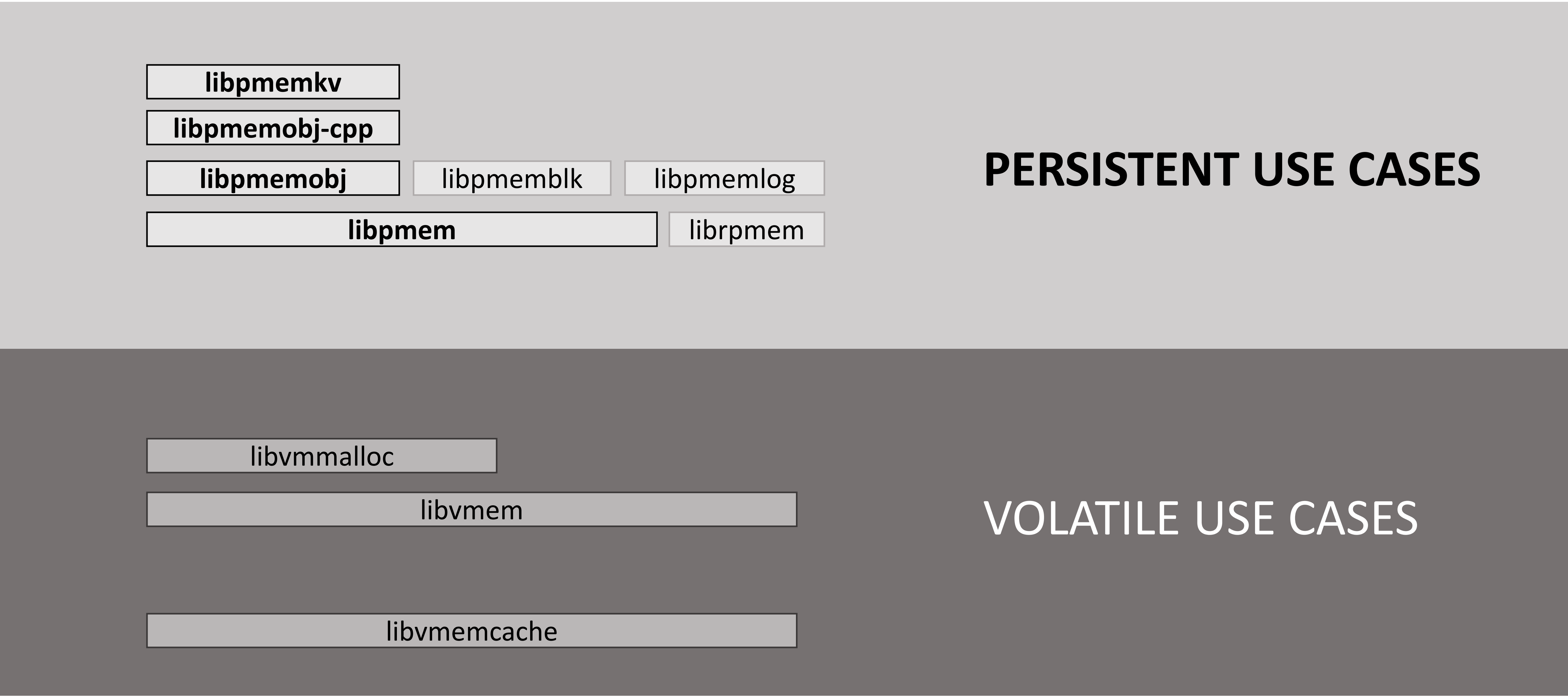
PMDK Primer

pmem.io
github.com/pmem/



PMDK Primer – focus of this presentation

pmem.io
github.com/pmem/



Usability

libpmemobj 0.0.1 API (December 2014)

```
struct node *
linked_list_insert(PMEMobjpool *pop, const char *str)
{
        struct base *bp = pmemobj_root_direct(pop, sizeof (*bp));
        jmp_buf env;

        if (setjmp(env) == 0) {
                /* try the transaction... */
                pmemobj_tx_begin_lock(pop, env, &bp->mutex);

                /* allocate the new node to be inserted */
                PMEMoid newoid = pmemobj_alloc(sizeof (struct node));
                struct node *newnode = pmemobj_direct_ntx(newoid);

                /* fill it in and link it in */
                newnode->str = pmemobj_strdup(str);
                newnode->next = bp->head;
                PMEMOBJ_SET(bp->head, newoid);

                pmemobj_tx_commit();

                return newnode;
        } else {
                /* transaction aborted */
                return NULL;
        }
}
```

```
struct base {
        PMEMoid head;
        PMEMmutex lock;
};

struct node {
        PMEMoid str;
        PMEMoid next;
};
```

- Very verbose
- Manual instrumentation
- No type information associated with objects
- What the hell is setjmp()?

libpmemobj 1.0 API (May 2016)

```
struct node *
linked_list_insert(PMEMobjpool *pop, const char *str)
{
        TOID(struct base) bp = pmemobj_root(pop, sizeof(struct base));
        struct node *ret = NULL;

        TX_BEGIN_PARAM(pop, TX_PARAM_MUTEX, &D_RW(bp)->lock, TX_PARAM_NONE) {
                TOID(struct node) newoid = TX_NEW(struct node);
                D_RW(newoid)->str = TX_STRDUP(str, 0);
                D_RW(newoid)->next = D_RO(bp)->head;

                TX_SET(bp, head, newoid);

                ret = D_RW(newoid);
        } TX_ONABORT {
                ret = NULL;
        } TX_END

        return ret;
}
```

```
struct base {
        TOID(struct node) head;
        PMEMmutex lock;
};

struct node {
        PMEMoid str;
        TOID(struct node) next;
};
```

- Slightly less verbose
- Manual instrumentation
- Some type information associated with objects
- Transparent transaction lifecycle

libpmemobj++ API (December 2016)

```
persistent_ptr<struct node>
linked_list_insert(pool_base &pop, std::string str)
{
        auto bp = pop.root();
        persistent_ptr<struct node> newoid;

        try {
                transaction::run(pop, [&] {
                        newoid = make_persistent<node>();

                        newoid->str = str;
                        newoid->next = bp->head;

                        bp->head = newoid;
                }, &bp->lock);
        } catch (pmem::transaction_error &) {
                newoid = nullptr;
        }

        return newoid;
}
```

```
struct base {
        persistent_ptr<struct node> head;
        pmem::obj::mutex lock;
};

struct node {
        pmem::obj::string str;
        persistent_ptr<struct node> next;
};
```

- Code nearly identical to volatile
- Minimal instrumentation
- Smart pointers, persistence as a type-associated property
- Closure transactions

Lesson 1

Persistence introduces a significant change to how programs are written, anything we can do to make it easier is a win.

- Don’t release too early – it might take a significant amount of time to get to something good.

- Types are great – it’s easier to add type qualifiers in a few places than it is to instruments large swaths of code.

- Metaprogramming helps – writing transactions in plain old C is… tricky. Adding macros helps, but using a language with proper metaprogramming capabilities makes coding a whole lot easier.

pmemkv 0.8 API (June 2019)

```
#include <libpmemkv.hpp>

using namespace pmem::kv;

int main()
{
        pmemkv_config *cfg = pmemkv_config_new();

        int ret = pmemkv_config_put_string(cfg, "path", PATH.c_str());
        ret = pmemkv_config_put_uint64(cfg, "size", SIZE);

        db *kv = new db();
        status s = kv->open("vsmap", cfg);

        s = kv->put("key1", "value1");

        std::string value;
        s = kv->get("key1", &value);

        kv->get_all([](string_view k, string_view v) {
                LOG(" visited: " << k.data()); return 0;
        });

        delete kv;
        return 0;
}
```

- Simple and familiar key-value store interface
- Completely abstracts away the storage
- Easy to adopt

Lesson 2

Better the devil you know…

We’ve observed more initial interest in intermediate PMEM adoption solutions such as pmemkv, normal file I/O or volatile approaches than in libpmemobj.

- A familiar, but constrained, API is significantly easier to adopt than an unknown one that fully takes advantage of PMEM.
- Specialized solutions (or less general purpose) provide a convenient starting point to persistent memory, allowing developers to progressively learn the stack – from simplest to the most complex.

Persistent memory allocator

Most research projects, and libpmemobj, try to use the same model of dynamic memory allocation that's used for volatile memory.

```
transaction::run(pop, [&] {
        root->object = make_persistent<mytype>();
});
```

PERSISTENT MEMORY

```
root->object = make_shared<mytype>();
```

VOLATILE MEMORY

From the API perspective this makes perfect sense, but…

what about fragmentation?

Lesson 3

Fragmentation in long-lived heaps can become a significant problem. Most traditional ways of hiding or dealing with this problem fall short of solving it.

- Defragmentation using virtual memory might eventually lead to a situation where there are no physically contiguous 4k pages, preventing the use of huge pages – which might be very important when taking advantage of large capacities of PMEM.
- Compaction algorithms might require excessive amount of CPU time and bandwidth (which is limited on PMEM) to properly do its job on terabyte heaps.
- Applications, if possible, should instrument the allocator with the information about size of its objects, thus allowing for optimal on-media arrangement.

Consistency in presence of failures

Constructing fail-safe algorithms is a non-trivial task. Even with libpmemobj and transactions it’s easy to make a mistake.

```
$ valgrind --tool=pmemcheck ./app

Number of stores not made persistent: 1
Stores not made persistent properly:
[0]    at 0x400794: main (example.c:7)
            Address: 0xfff000124  size: 4 state: DIRTY
Total memory not made persistent: 4
```

```
$ pmreorder ./app

WARNING:pmreorder:File /tmp/test_ex_pmreorder1/testfile inconsistent
WARNING:pmreorder:Call trace:
Store [0]:
    by  0x400CDB: list_insert_inconsistent (pmreorder_list.c:144)
    by  0x400E84: main (pmreorder_list.c:185)
```

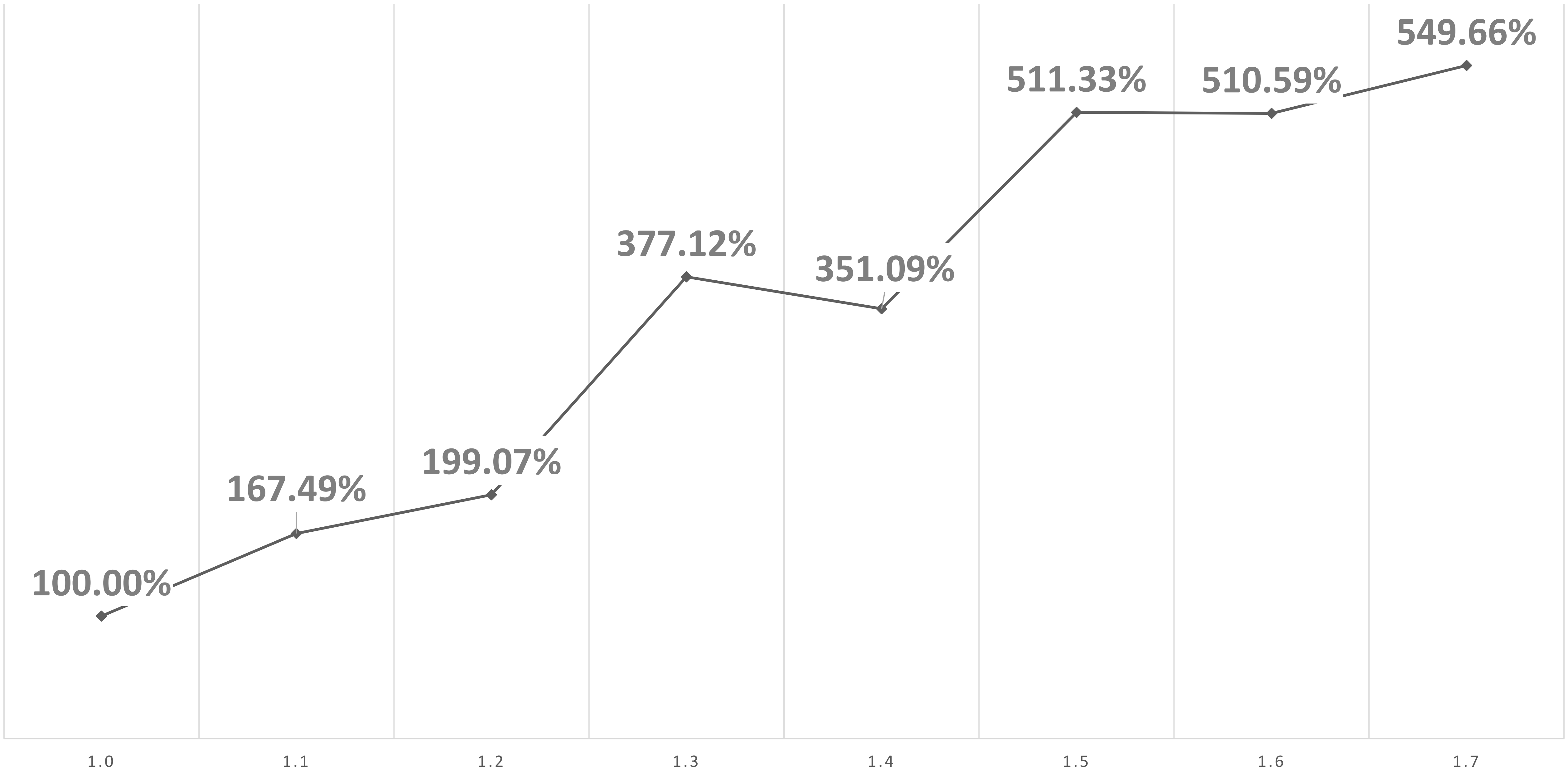
Lesson 4

PMEM programming *is* hard.

- Testing through killing an application or pulling the plug on a server, while useful, does not produce deterministic results and is difficult to debug.
- Define data structure invariants, then verify early and often.
- It might get computentially prohibitive to exhaustively check all possible memory states an application can be in. So instead of doing that, divide and conquer - Isolating problems into smaller bits helps reduce the amount of possible memory states, but also makes reasoning about consistency easier.

Performance

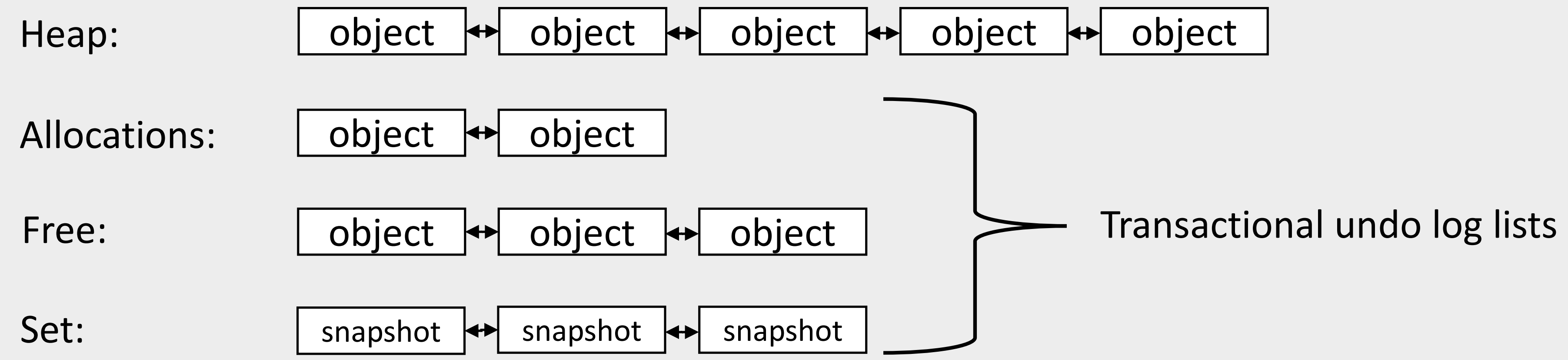
LIBPMEMOBJ RELATIVE PERFORMANCE ACROSS VERSIONS (B-TREE BENCHMARK)



Implementing persistent transactions

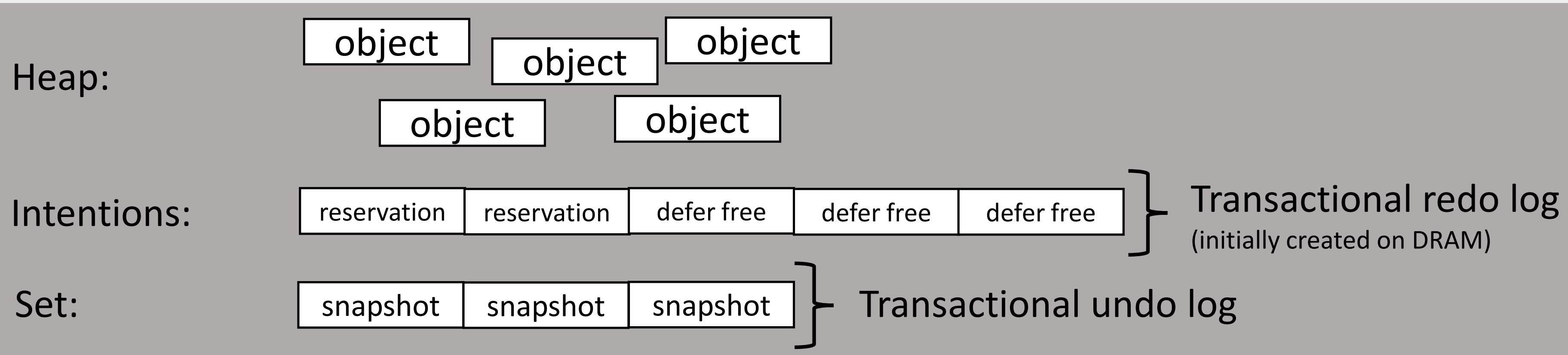
Since its first version, libpmemobj had three different implementation of undo logs, and at least once almost the entire transactions module was rewritten.

libpmemobj 1.0



- Doubly linked lists of objects
- Easy to reason about…
- … but very slow

libpmemobj 1.5



- No lists whatsoever
- Hybrid redo/undo transactions
- Minimal overhead
- Non-committed actions don't use PMEM (allocs, free)

Lesson 5

Accessing PMEM is expensive compared to DRAM. Ideas that sound reasonable for DRAM latencies might not work well for Persistent Memory.

- Individual cache-misses matter, selectively caching variables in DRAM is often a worthwhile strategy.
- Read bandwidth is higher than write bandwidth. Trading reads for reduced number of writes makes sense in many scenarios.
- With Persistent Memory* disrupting the memory hierarchy, cache oblivious data structures become even more beneficial.

* Current generation of Intel® Optane™ DC Persistent Memory uses 256 byte ECC blocks.

Copying data to PMEM

memcpy() isn’t enough when it comes to writing performant Persistent Memory enabled code.

```
void *memcpy(void * destination, const void * source, size_t num);
```

```
void *pmem_memcpy_persist(void *pmemdest, const void *src, size_t len);
```

```
void *pmem_memcpy(void *pmemdest, const void *src, size_t len, unsigned flags);
flags = PMEM_F_MEM_NODRAIN | PMEM_F_MEM_NOFLUSH | PMEM_F_MEM_NONTEMPORAL | PMEM_F_MEM_TEMPORAL
```

Since the initial version of libpmem, we’ve created two implementations of PMEM optimized memcpy. And it’s not because our code can potentially be faster than libc memcpy (it might not). It’s to avoid flushing and unnecessary store fences.

Lesson 6

Non-temporal stores are crucial to achieving decent performance of persistent memory algorithms.

- Using memcpy implementation that is deterministic w.r.t. usage of NT stores allows application to avoid cache flushing when cache was bypassed.
- Cache-line alignment is doubly important. Aligned non-temporal stores can entirely avoid cache misses that would normally happen with temporal copies.
- Delaying store fence, normally required after memcpy() with NT instructions, until data is actually required to reach persistent memory can yield non-trivial performance improvements, especially when multiple discontiguous copies are being made.

Persistent memory allocator - again

The second most modified module of libpmemobj, after transactions, is the allocator. It's performance is crucial to all parts of the library.

```
int pmalloc_construct(PMEMobjpool *pop, uint64_t *off, size_t size,
        pmalloc_constr constructor, void *arg);
```

libpmemobj 1.0

```
int palloc_reserve(struct palloc_heap *heap, size_t size, palloc_constr constructor, void *arg,
        uint64_t extra_field, uint16_t object_flags, uint16_t class_id,
        uint16_t arena_id, struct pobj_action *act);
void palloc_publish(struct palloc_heap *heap, struct pobj_action *actv, size_t actvcnt,
        struct operation_context *ctx);
```

libpmemobj 1.3

One of the biggest changes we've made to it is departure from immediate persistence model to a delayed one. This means that persistent metadata is no longer immediately modified to reflect a heap operation, and instead objects are reserved from purely volatile (held in DRAM) state and persistent metadata modifications are batched. This led to 64x reduction in metadata modifications for transactional heap operations in select scenarios, amortizing the cost.

Lesson 7

Hybrid DRAM/PMEM algorithms tend to have the lowest latency.

- Staging and coalescing PMEM changes in DRAM might be a good idea when the algorithm repeatedly writes to the same persistent memory location.
- Preparing data in DRAM and offloading it asynchronously to PMEM might be beneficial when delaying the main thread is undesirable (e.g., reading data from a socket to pmem, and processing it in another thread).
- Storage-focused data structures (Fractal Trees, LSM) also make sense for PMEM when modified to take its properties into account.

Thank you!

Questions?