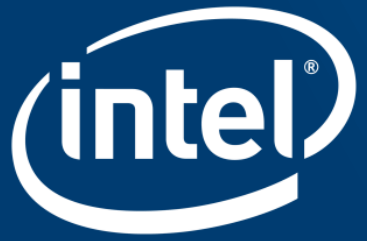
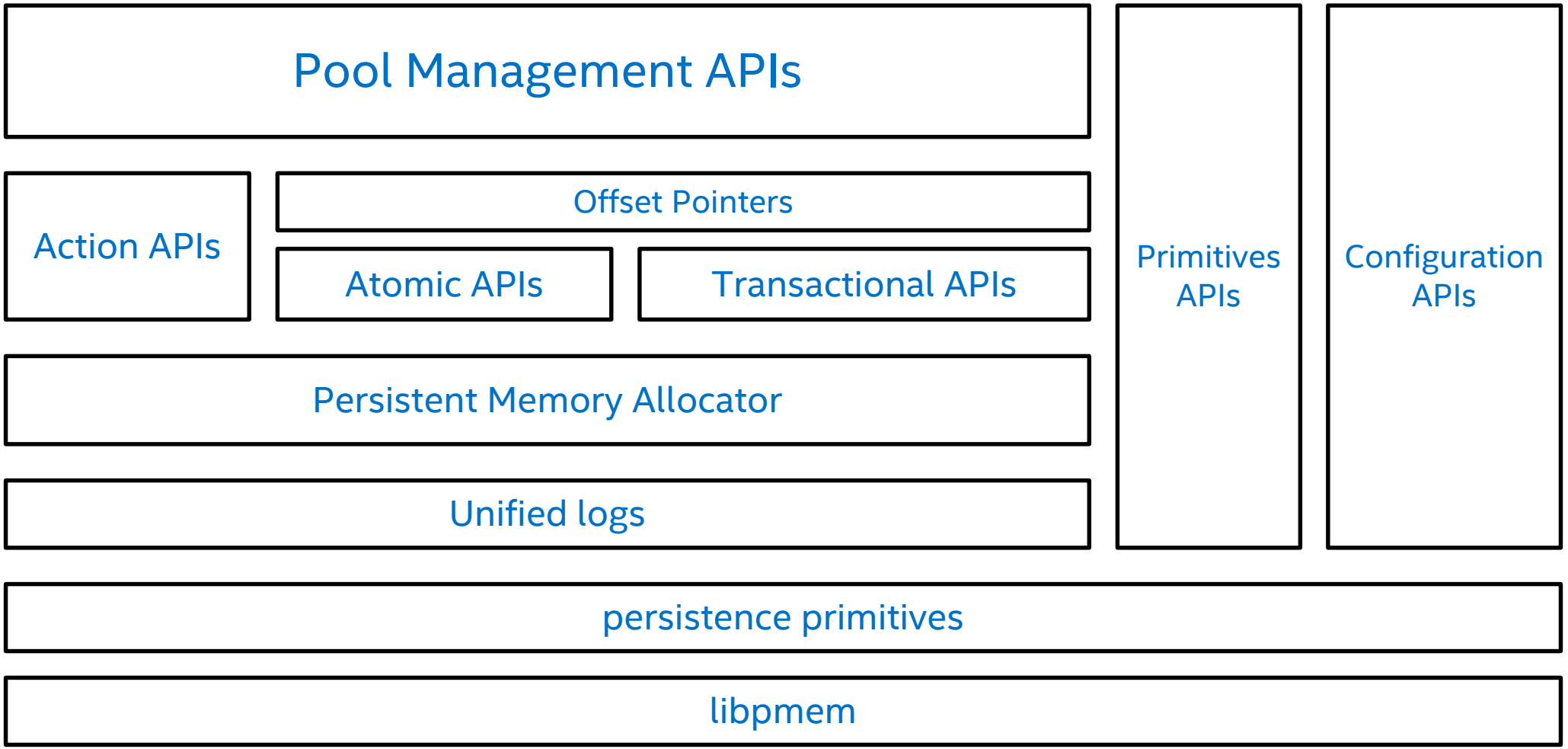


LIBPMEMOBJ DEEP DIVE

Piotr Balcer
Intel® Data Center Group

libpmemobj overview



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

Redo logging

Address	Size	New Data
0x4000FF00	8	12345
0x40ADFF00	8	54321
Finish flag		

- 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

Undo logging

Address	Size	New Data
0x4000FF00	8	12345
0x40ADFF00	8	54321

- 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

Persistent Memory Allocator

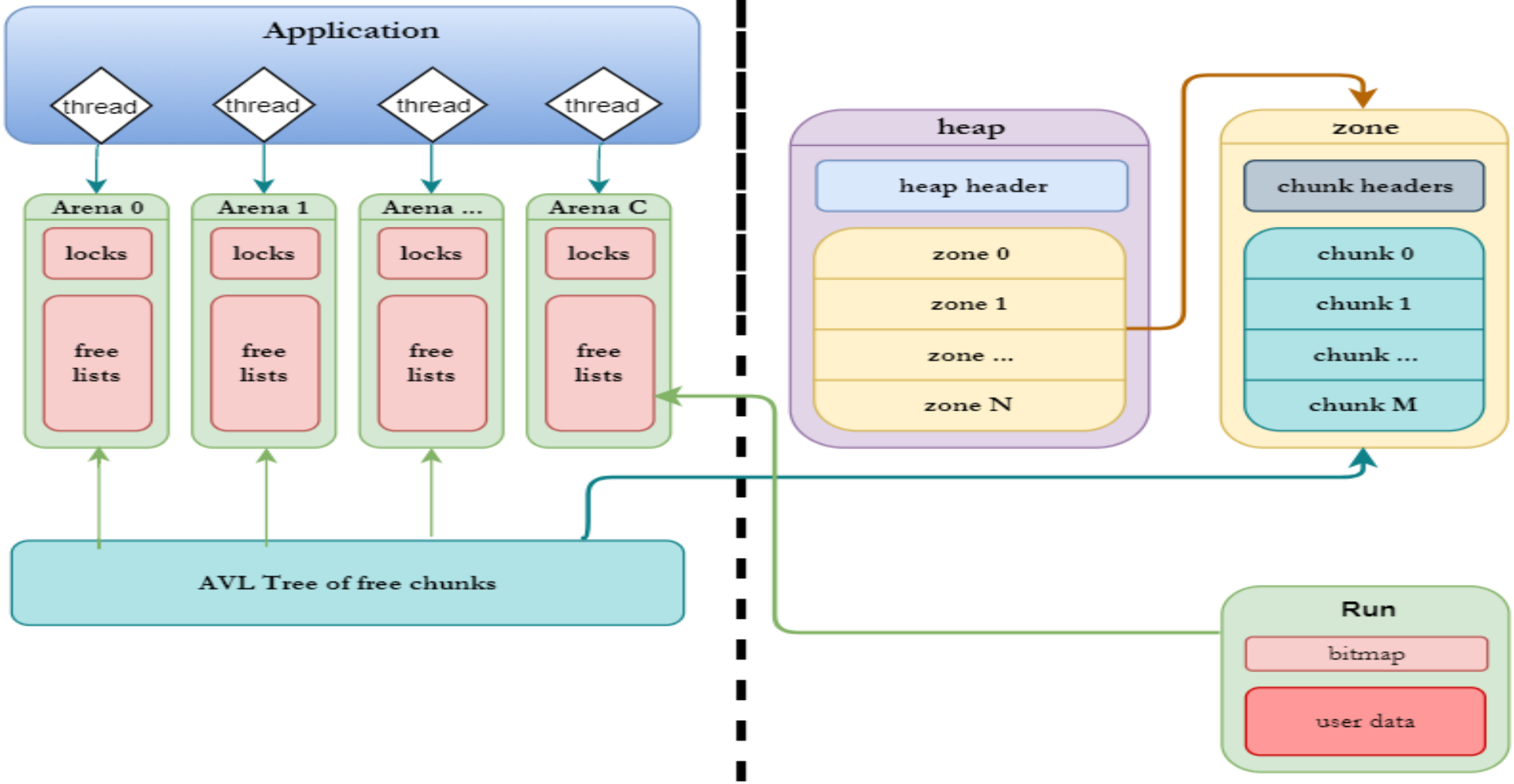
- Implemented from scratch for libpmemobj
- It maintains runtime and persistent state of the heap
  - Runtime state is used for fast allocation
  - Persistent state is used for durability
    - Small bitmaps for small allocations
    - Persistent Boundary tags for large allocations
- Uses segregated free-lists for small allocations and best-fit using AVL Tree for large allocations. These data structures are runtime.

Transient State

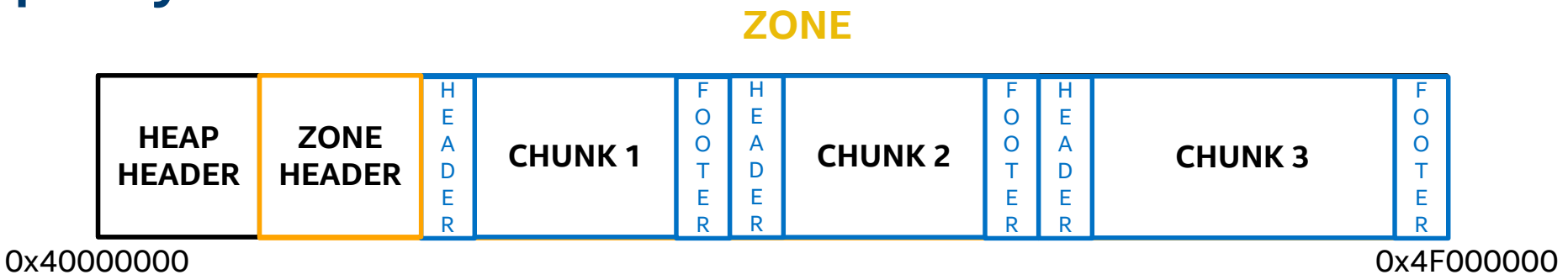
Allocator's runtime data structures which are allocated from normal DRAM for performance reasons.

Persistent State

Allocator's on-media data structures which are kept on non-volatile memory and are updated in fail-safe atomic way.



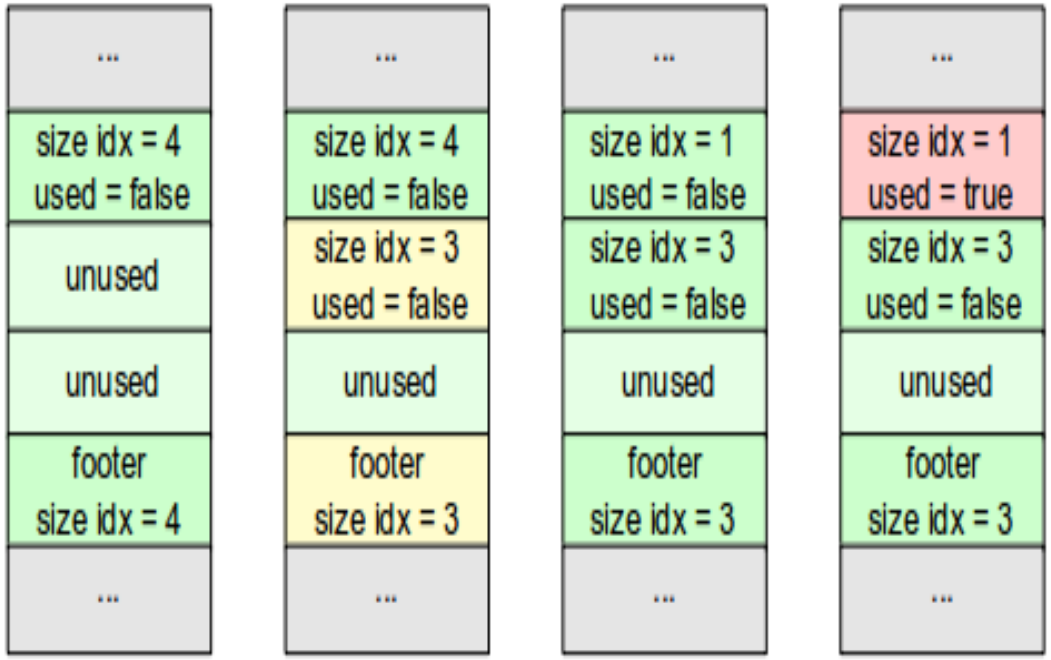
Heap layout



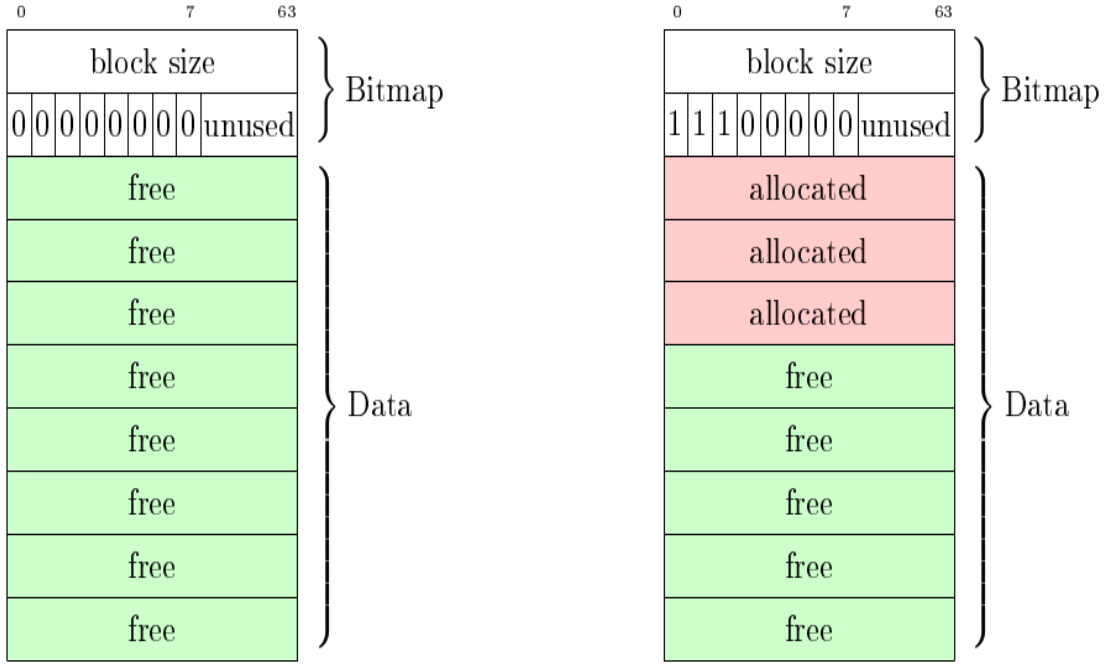

```
struct chunk {
        uint8_t data[CHUNKSIZE];
};

struct chunk_run_header {
        uint64_t block_size;
        uint64_t alignment;
};
```

```
struct chunk_run {
        struct chunk_run_header hdr;
        /* bitmap + data */
        uint8_t content[RUN_CONTENT_SIZE];
};

struct chunk_header {
        uint16_t type;
        uint16_t flags;
        uint32_t size_idx;
};
```

Example of Metadata Changes during Allocation

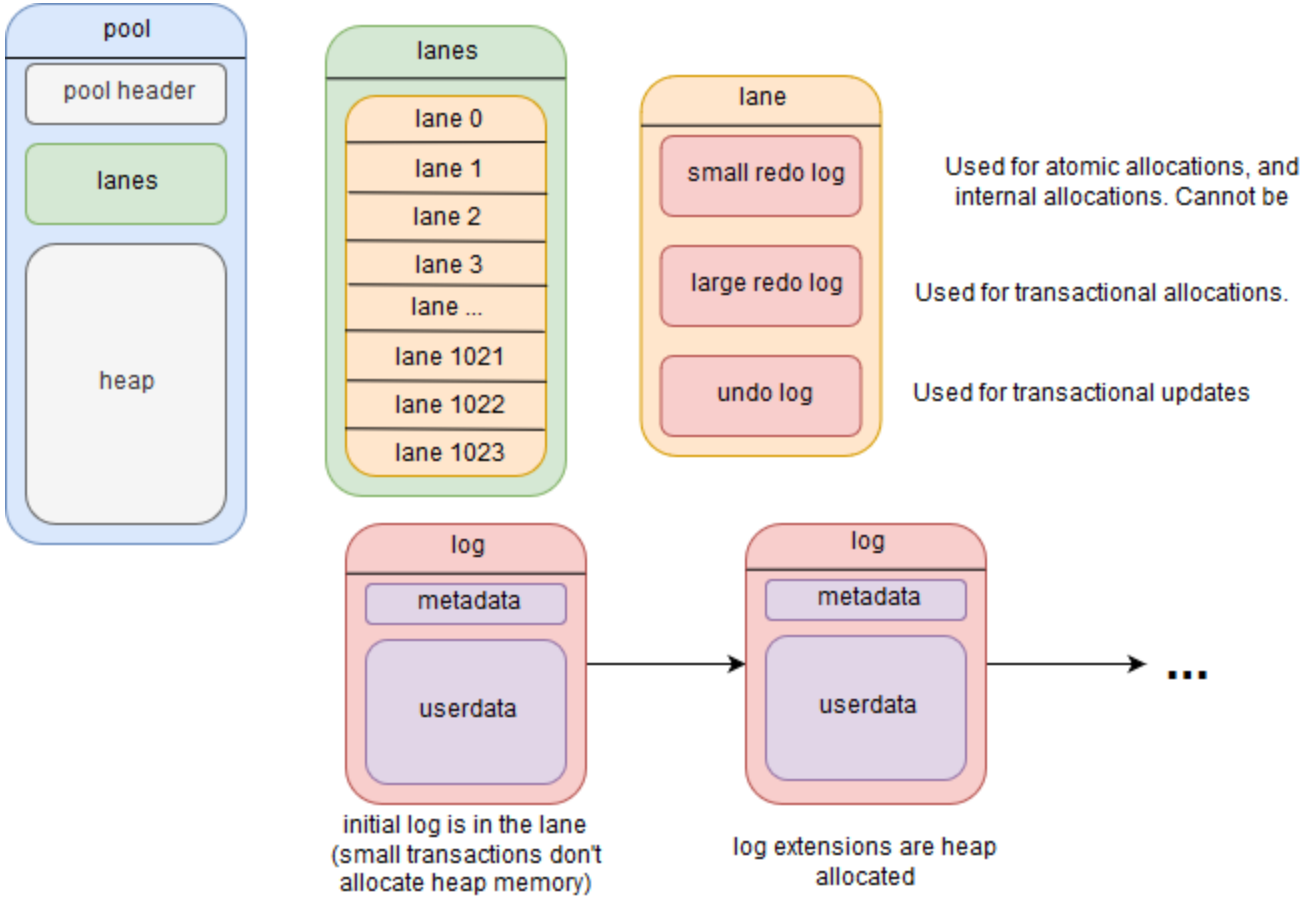


Changes in the metadata in Chunk headers before and after allocation of 1 Chunk

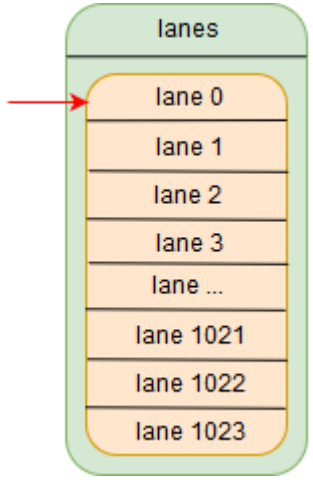


Changes in the metadata for Smaller allocations (Runs)

on-media data structures



```
transaction::run(pop, [&] {
        auto n = make_persistent<pmem_entry>();

        n->value = value;
        n->next = nullptr;

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




```
struct lane_layout {
  /*
   * Redo log for self-contained and 'one-shot' allocator operations.
   * Cannot be extended.
   */
  struct ULOG(LANE_REDO_INTERNAL_SIZE) internal;
  /*
   * Redo log for large operations/transactions.
   * Can be extended by the use of internal ulog.
   */
  struct ULOG(LANE_REDO_EXTERNAL_SIZE) external;
  /*
   * Undo log for snapshots done in a transaction.
   * Can be extended/shrunk by the use of internal ulog.
   */
  struct ULOG(LANE_UNDO_SIZE) undo;
};
```

```
#0  lane_hold(pop = 0x7ffff5200000, lanep = 0x7ffff7fbf7f0) at lane.c:545
#1  0x00007ffff7bb1f92 in pmemobj_tx_begin(pop = 0x7ffff5200000, env = 0x0)
                                    at tx.c:684
#2  0x0409a62 in pmem::obj::transaction::run<>(pmem::obj::pool_base&, std::function<void()>)
                                    at transaction.hpp:402
#3  0x0408f61 in examples::pmem_queue::push(this = 0x7ffff55c0550, pop = ..., value = 1)
                                    at queue.cpp:118
#4  0x0408582 in main(argc = 4, argv = 0x7fe0d8) at queue.cpp:199
```

Starting a transaction initializes transaction runtime state and grabs a unique lane for persistent metadata

```
transaction::run(pop, [&] {
        auto n = make_persistent<pmem_entry>();

        n->value = value;
        n->next = nullptr;

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

```
static __thread struct tx tx;
struct tx {
  VEC(, struct pobj_action) actions;
  ..
};
```

```
palloc_reserve(&pop->heap, size, constructor, &args, type_num, 0, CLASS_ID_FROM_FLAG(args.flags), action);
```

Allocating a new object only *reserves* a new memory block in a runtime state of the allocator
No allocator metadata is permanently changed.

```
transaction::run(pop, [&] {
        auto n = make_persistent<pmem_entry>();

        n->value = value;
        n->next = nullptr;

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

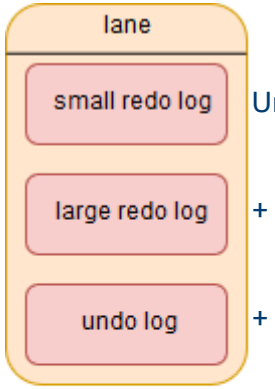
```
struct lane_layout {
  /*
   * Redo log for self-contained and 'one-shot' allocator operations.
   * Cannot be extended.
   */
  struct ULOG(LANE_REDO_INTERNAL_SIZE) internal;
  /*
   * Redo log for large operations/transactions.
   * Can be extended by the use of internal ulog.
   */
  struct ULOG(LANE_REDO_EXTERNAL_SIZE) external;
  /*
   * Undo log for snapshots done in a transaction.
   * Can be extended/shrunk by the use of internal ulog.
   */
  struct ULOG(LANE_UNDO_SIZE) undo;
};
```

```
struct log {
  /* 64 bytes of metadata */
  uint64_t checksum; /* checksum of ulog header and its entries */
  uint64_t next; /* offset of ulog extension */
  uint64_t capacity; /* capacity of this ulog in bytes */
  uint64_t unused[5]; /* must be 0 */
  uint8_t data[capacity_bytes]; /* N bytes of data */
}
```

```
struct ulog_entry_base {
  uint64_t offset = (uintptr_t)pop - (uintptr_t)this->head;
};

/*
 * ulog_entry_buf - ulog buffer entry
 */
struct ulog_entry_buf {
  struct ulog_entry_base base; /* offset with operation type flag */
  uint64_t checksum = checksum(ulog_entry_buf);
  uint64_t size = sizeof(this->head)
  uint8_t data[] = memcpy(data, n, sizeof(n));
};
```

Taking a snapshot creates a new ulog entry in the undo log.

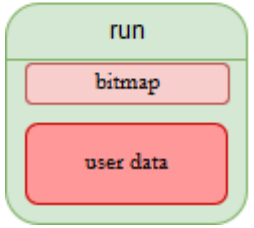
```
transaction::run(pop, [&] {
        auto n = make_persistent<pmem_entry>();

        n->value = value;
        n->next = nullptr;

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



Unused because this is a small transaction and did not need extended logs

+ persistent allocation redo log entry

+ persistent snapshots in undo logs




```
struct ulog_entry_val {
  uint64_t offset; = &run->bitmap
  uint64_t value;  |= 0b00010000
};
```

Finishing a transaction:

1. Creates and processes a redo log for all allocations (palloc_publish)
2. Removes snapshots from undo logs
3. Releases the lane

Summary

- Memory Management is a crucial building block in building an efficient Allocator
- In this presentation we showed:
  - Allocator's Persistent Heap & Volatile Memory layout
  - Fail-safe atomic and transactional allocations
  - Allocator's recovery mechanism

Q&A