Lessons Learned from MemVerge

Yue Li and Wei Kang
MemVerge

09/06/2019

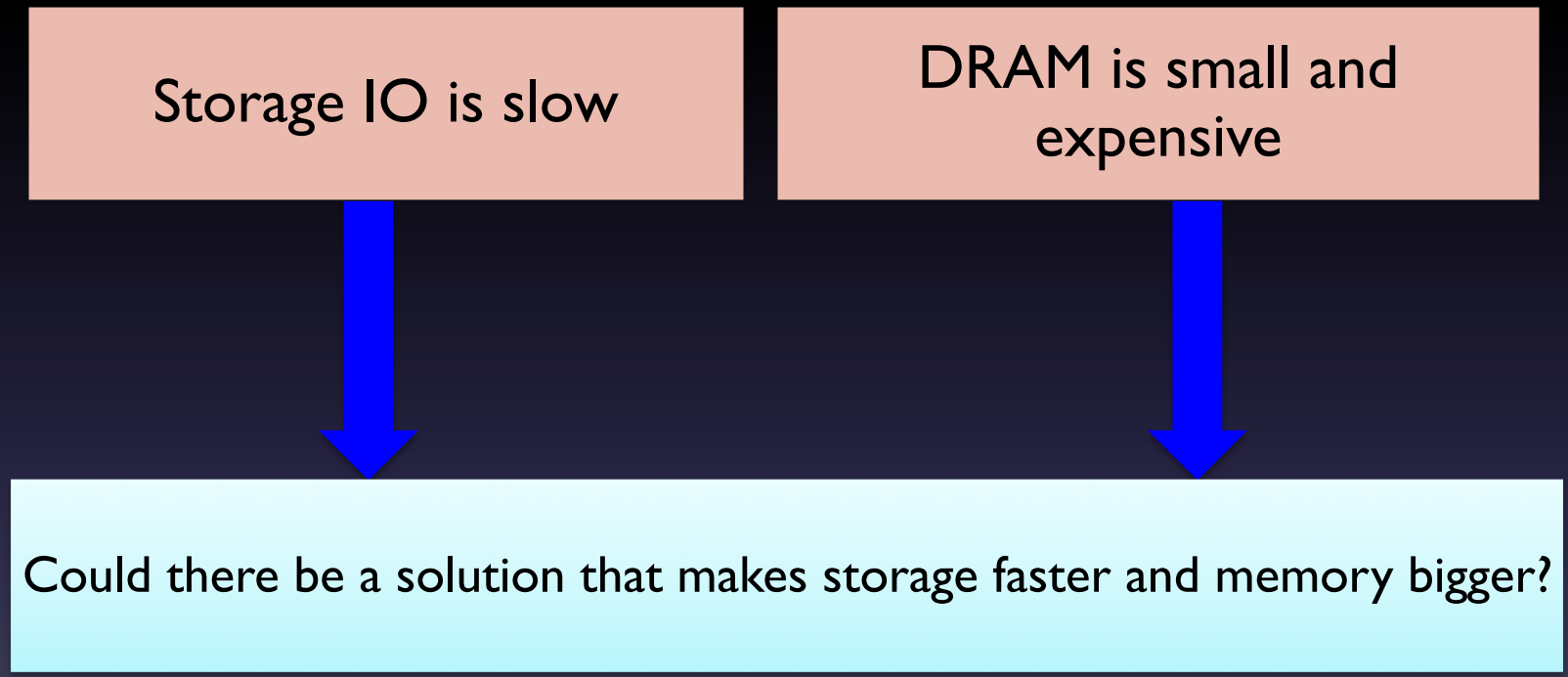
Agenda

- MemVerge Overview
- Our PMDK experience
  - libpmemobj
    - Memory allocation
    - Concurrent memory allocation
- Summary

About MemVerge

- Founded in 2017
- Headquartered in Milpitas, California, USA
- R&D teams in US and China
- Focus: next generation data infrastructure powered by persistent memory
  - Current product: MemVerge Distributed Memory Objects (DMO)
  - Beta version is now available for early PoC customers.
  - https://www.memverge.com/#beta

Data Infrastructure Pain Points

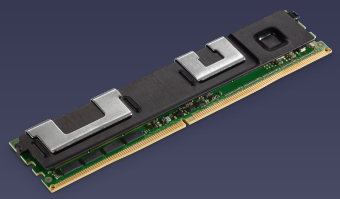


A New Class of Memory

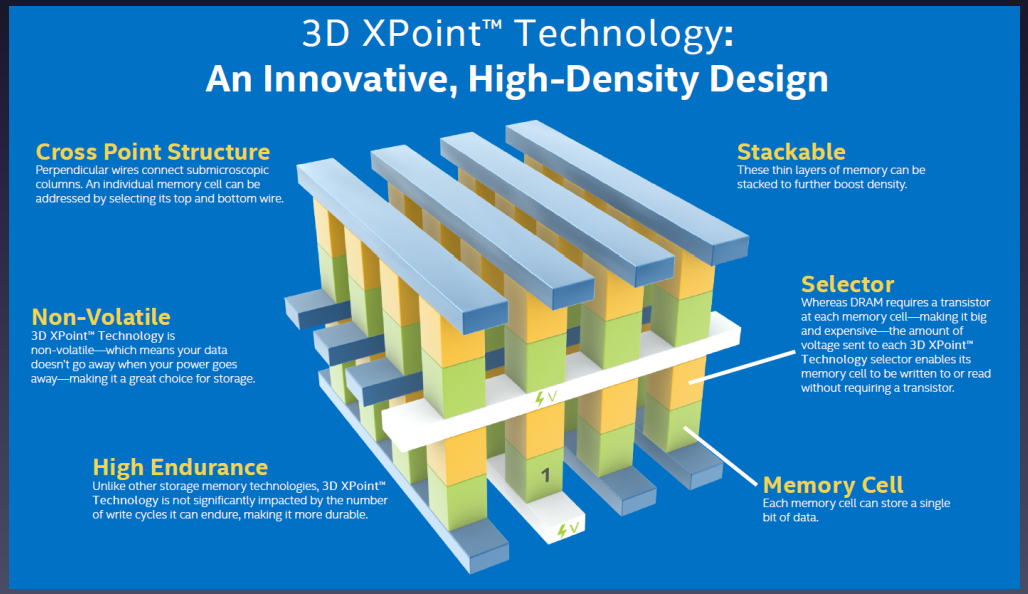
- Persistent Memory (aka Storage Class Memory, NVRAM)
  - Low latency and high throughput, near the speed of DRAM
  - High density and non-volatility, like NAND flash

- Intel/Micron 3D XPoint
  - NVMe SSD (Optane SSD)
  - Byte-addressable DIMM (Optane DC PMEM)

SSD

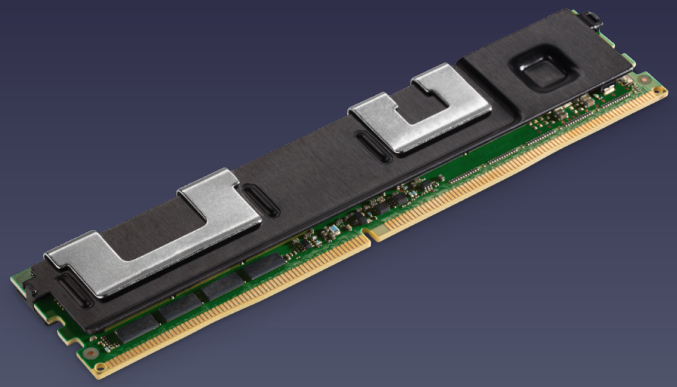


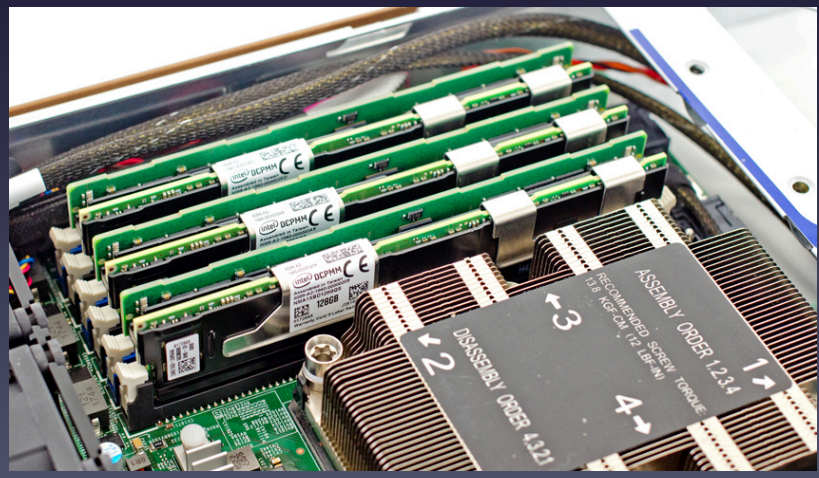
PMEM DIMM



Storage Class Memory has Emerged!

- Intel Delivered Optane DC Persistent Memory based on 3D XPoint tech in Q2 2019
  - Revenue projected to reach $3.6B by 2023
- Additional major vendors to join the foray by 2022
  - Making this a $10B+ market by 2025
- Software ecosystem will be key for technology adoption
  - Identifying work load with strong ROI
  - Allowing adoption without requiring application rewrite

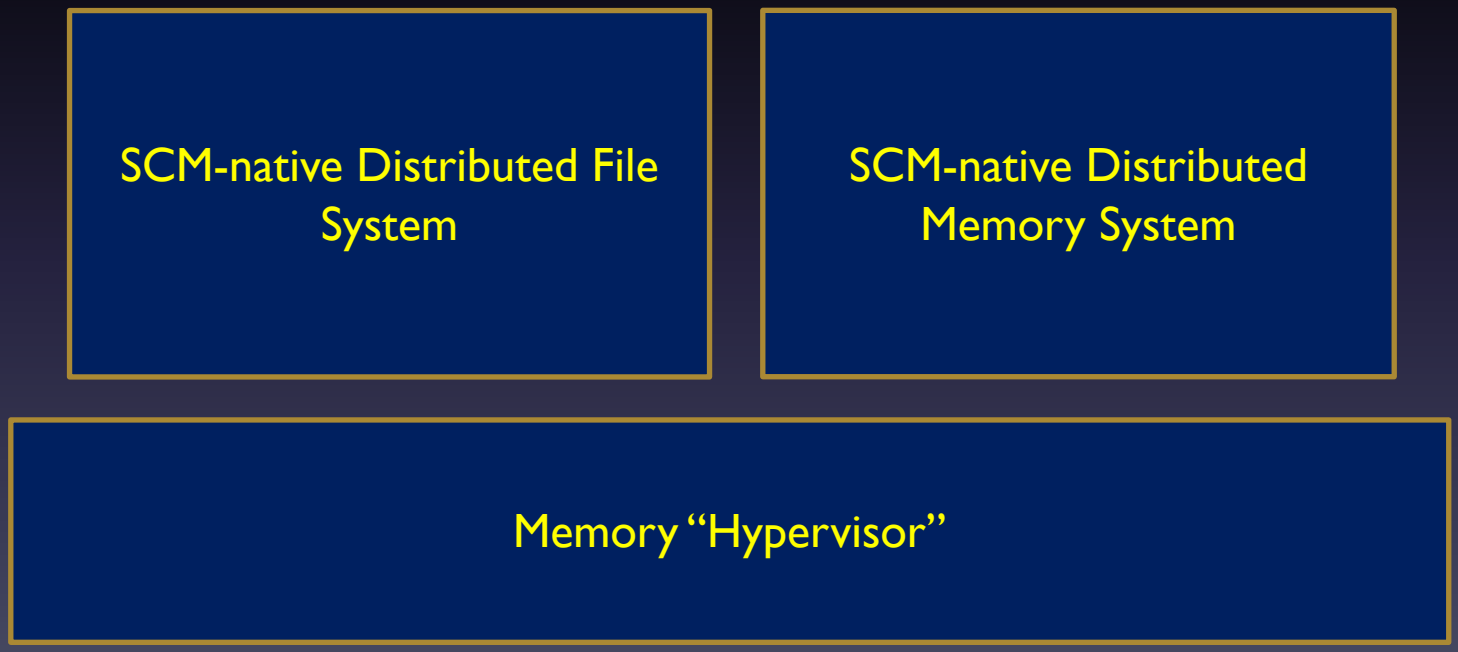




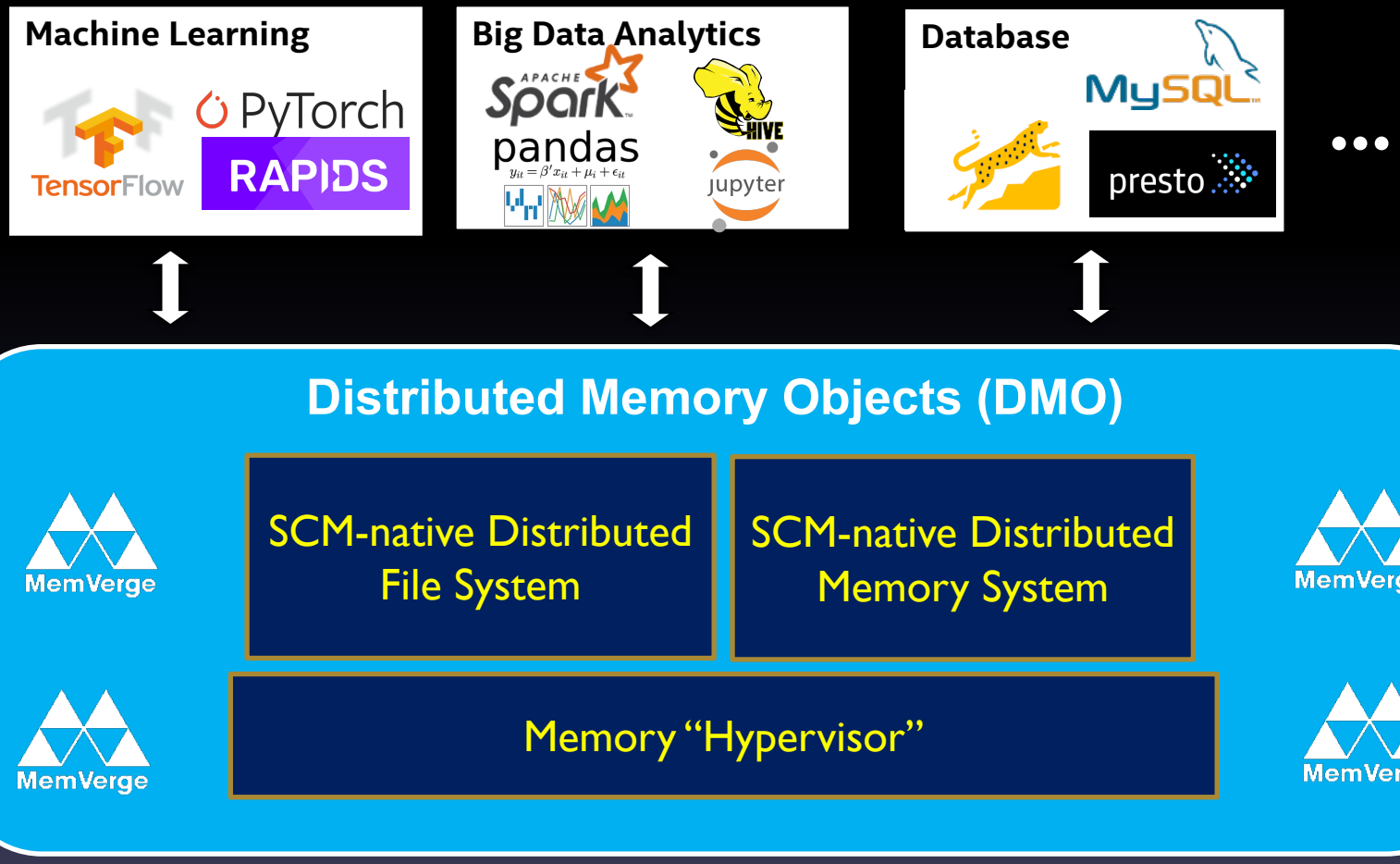
Picture source: storagereview.com

World's First
Memory Converged Infrastructure

MemVerge software leverages Storage Class Memory technology to deliver larger memory and faster storage to applications without requiring application rewrites



Memory Converged Infrastructure Platform



MemVerge is an Avid PMDK User

- This presentation
  - Share our stories as PMDK user
    - libpmemobj
    - Focus on memory allocation
    - Related Intel webinar on PMDK allocator

      https://software.intel.com/en-us/videos/introduction-to-persistent-memory-allocator-and-transactions

Developing with PMEM

- Native support by recent Linux releases
  - e.g. CentOS/RHEL 7.5+ with vanilla kernel
- Great if real PMEM presents
  - e.g. Intel Optane DC PMEM
- PMEM emulation is easy
  - Augmenting grub with `memmap=2G!4G`
  - Works well on VMs, commodity laptops…
  - Emulated PMEM device survives across powered reboot

PMDK Strengths

- PMEM-native data access
  - `pmem_persist` and `pmem_memcpy`
  - Automatically applies platform-dependent optimization on accessing PMEM
    - Avoid your own tedious and error-prone implementations
- Transactions are readily available (for atomicity)
- Versatile
  - Different allocators (libpmemlog, libpmemblk, libpmemobj and libvmmaloc)
  - Different data abstractions (raw memory, object store, block, and key-value)
    - Support different drivers including pmem block device and device DAX

libpmemobj

- Allow objects get allocated and persisted in PMEM
- Simple to use
  - Create a pmem pool with `pmempool` utility, and we are ready to go
- Allocator has a good memory utilization across a wide range of allocation sizes
  - Great for 16K and 64K
  - Can be better for 1MB+

```
int main(int argc, char *argv[]) {
  PMEMobjpool *pop = pmemobj_open("/dev/dax0.0", "mvdmo");
  if (pop == NULL || argc < 2) {
    return 1;
  }
  uint64_t n = 0;
  uint32_t allocation_size = std::stoul(argv[1]);
  for (;;) {
    PMEMoid oid;
    int rc = pmemobj_alloc(pop, &oid, allocation_size, 1, nullptr, nullptr);
    if (rc)
      break;
    ++n;
  }
  uint64_t totsize = 4225761280ULL;
  uint64_t utilized = n * allocation_size;
  std::cout << totsize << "," << utilized << "," << totsize - utilized << " : "
      << utilized * 1000 / totsize << std::endl;
  pmemobj_close(pop);
  return 0;
}
```

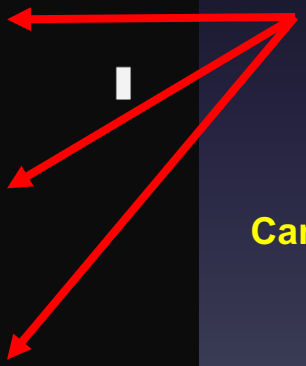
Allocation Size	Utilization %
512	87.2
4096	92.1
16384	97.4
65536	98.2
262144	93.6
1048576	74.8
2097152	88.7

*Assume using 4GB pool

Understand the Results via `pmempool info`

- Helps us understand our utilization results

```
Major                   : 1
Minor                   : 0
Chunk size              : 262144
Chunks per zone         : 65528
Checksum                : 0x4bf97ca8f7542425 [OK]

Zone 0:

 Chunk                  : 0
 Type                   : used
 Flags                  : 0x1 compact header
 Size idx               : 9

 Chunk                  : 9
 Type                   : used
 Flags                  : 0x1 compact header
 Size idx               : 9

 Chunk                  : 18
 Type                   : used
 Flags                  : 0x1 compact header
 Size idx               : 9

 Chunk                  : 27
 Type                   : used
```



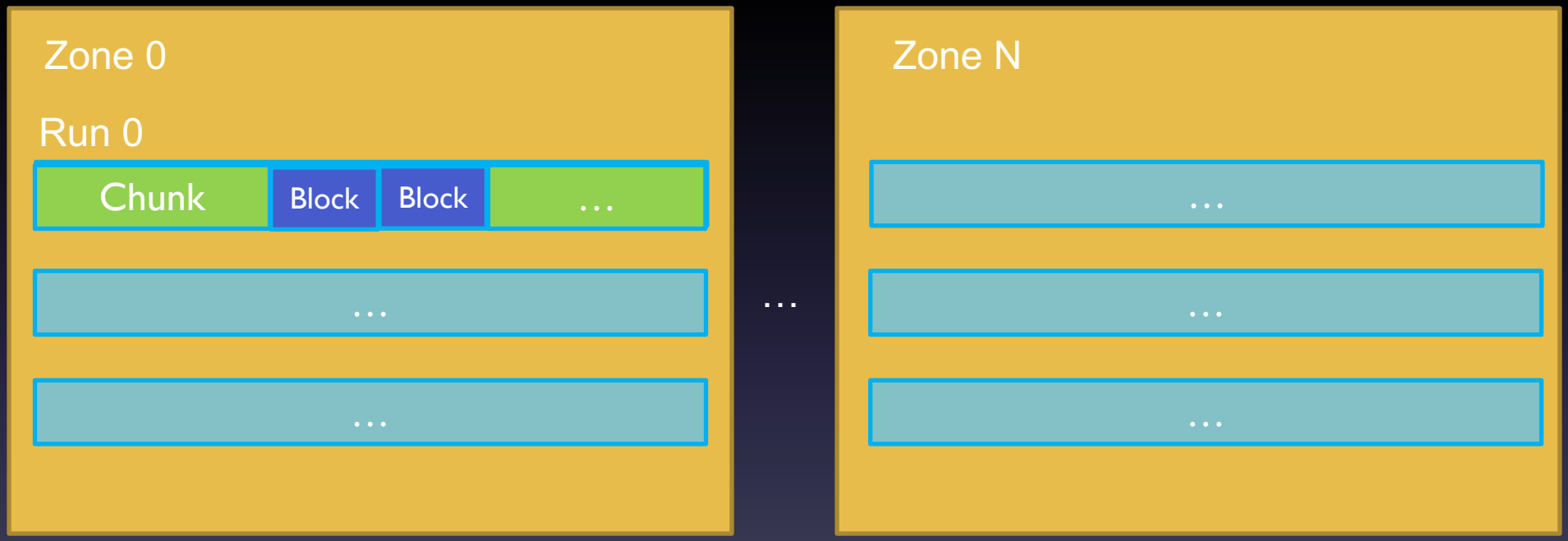
9 contiguous chunks per 2MB object

Utilization = 2MB / (9 * 256KB) = 88.7%!

Can we just use 8 chunks? (100% utilization)

*Output for 2MB allocations

libpmemobj
Persistent Heap Organization



pmempool info -H -C -b

- Enables heap, chunk and bitmap information

```
Chunk size               : 262144
Chunks per zone          : 65528
Checksum                 : 0x4bf97ca8f7542425 [OK]

Zone 0:

 Chunk                   : 0
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......

 Chunk                   : 16
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......

 Chunk                   : 32
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......
```

- 1 MB allocation
- 16 x 256KB contiguous chunks to create a “run” type area.
- Suballocated via bitmap.
- Only 3 x 1MB allocations can be made from the 16 x 256KB area
  - which ideally could fit four.
  - Due to overhead
  - Explains 74.8 % utilization

How to Improve Utilization

- Custom allocation classes
  - Let you specify
    - Allocation unit size
    - Number of allocation units
    - Whether allocation unit has a header and what type
    - The alignment of the units
  - libpmemobj has 53 pre-defined allocation classes
    - Allocator makes decision on which ones to use
  - Define additional allocation classes
    - Via `pmemobj_ctl_set`
  - Allocate through specific allocation classes
    - Function `pmemobj_xalloc()` can be parameterized by allocation class

Custom Allocation Class Example

- Allocation class has a name, following a naming convention

```
static void dump_alloc_classes(PMEMobjpool *pop) {
  char cls_desc[32];
  struct pobj_alloc_class_desc cls;
  for (int i = 0; i < 255; i++) {
    snprintf(cls_desc, sizeof(cls_desc), "heap.alloc_class.%d.desc", i);
    if (pmemobj_ctl_get(pop, cls_desc, &cls) >= 0) {
      printf("%s %lu\n", cls_desc, cls.alignment);
      printf("cls.unit_size: %lu\n", cls.unit_size);
      printf("cls.units_per_block: %d\n", cls.units_per_block);
      printf("cls.class_id: %d\n", cls.class_id);
    }
  }
}
```

`pmemobj_ctl_get` can retrieve the allocation class.

```
heap.alloc_class.0.desc 0
cls.unit_size: 262144
cls.units_per_block: 0
cls.class_id: 0
heap.alloc_class.1.desc 0
cls.unit_size: 128
cls.units_per_block: 2045
cls.class_id: 1
heap.alloc_class.2.desc 0
cls.unit_size: 192
cls.units_per_block: 1364
cls.class_id: 2

...


heap.alloc_class.51.desc 0
cls.unit_size: 246720
cls.units_per_block: 17
cls.class_id: 51
heap.alloc_class.52.desc 0
cls.unit_size: 262080
cls.units_per_block: 16
cls.class_id: 52
heap.alloc_class.53.desc 0
cls.unit_size: 349440
cls.units_per_block: 12
cls.class_id: 53
```

Custom Allocation Class to Improve Utilization

- Class 54 is the first unused class
- Let's use it for an efficient 1MB allocation
- 4KB alignment (page aligned)
- No header
- 256 units per run

- Use `pmemobj_xalloc()`
  - with `POBJ_CLASS_ID()` option

```
static int custom_alloc_class(PMEMobjpool *pop, size_t unit_size) {
  pobj_alloc_class_desc cls;
  cls.alignment = 4096;
  cls.header_type = POBJ_HEADER_NONE;
  cls.unit_size = unit_size;
  cls.units_per_block = 256;
  cls.class_id = 54; /* the first class id available after the pre-defined classes */

  if (pmemobj_ctl_set(pop, "heap.alloc_class.new.desc", &cls) < 0) {
    printf("Failed to create allocate class for unit size: %lu\n", cls.unit_size);
    return -1;
  }
  return cls.class_id;


int main(int argc, char *argv[]) {
  PMEMobjpool *pop = pmemobj_open("/dev/dax0.0", "mvdmo");
  if (pop == NULL) {
    return 1;
  }
  uint32_t allocation_size = 1024 * 1024;
  int clsid = custom_alloc_class(pop, allocation_size);
  if (clsid < 0) {
    pmemobj_close(pop);
    return -1;
  }

  uint64_t n = 0;
  for (;;) {
    PMEMoid oid;
    int rc = pmemobj_xalloc(pop, &oid, allocation_size, 1, POBJ_CLASS_ID(54), nullptr, nullptr);
    if (rc)
      break;
    ++n;
  }
  uint64_t totsize = 4225761280ULL;
  uint64_t utilized = n * allocation_size;
  std::cout << totsize << "," << utilized << "," << totsize - utilized << " : "
      << utilized * 1000 / totsize << std::endl;
  pmemobj_close(pop);
  return 0;
```

Utilization becomes 95% for 1MB allocation

Before: using default allocation class

```
Chunk size               : 262144
Chunks per zone          : 65528
Checksum                 : 0x4bf97ca8f7542425 [OK]

Zone 0:

 Chunk                   : 0
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......

 Chunk                   : 16
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......

 Chunk                   : 32
 Type                    : run
 Flags                   : 0x9 compact header
 Size idx                : 16
 Block size              : 135296
 Bitmap                  : 24 / 31
 xxxxxxxx xxxxxxxx xxxxxxxx .......
```

After: using custom allocation class

```
Chunk size               : 262144
Chunks per zone          : 65528
Checksum                 : 0x4bf97ca8f7542425 [OK]

Zone 0:

 Chunk                   : 0
 Type                    : run
 Flags                   : 0xe header none
 Size idx                : 1025
 Block size              : 1048576
 Bitmap                  : 255 / 255
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxx

 Chunk                   : 1025
 Type                    : run
 Flags                   : 0xe header none
 Size idx                : 1025
 Block size              : 1048576
 Bitmap                  : 255 / 255
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxx
```

Tips for Custom Allocation Class

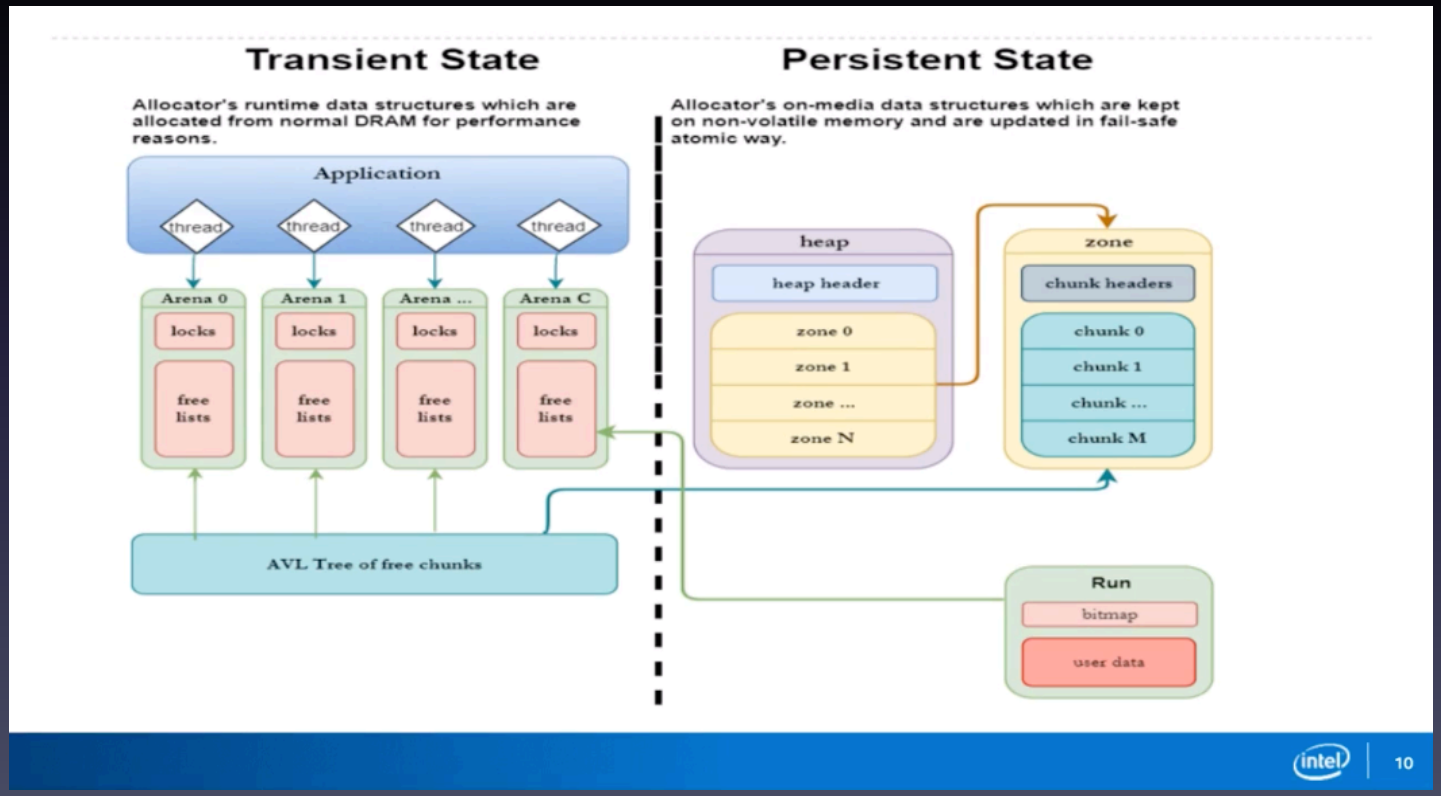
- Dramatically improves pool utilization
  - Especially when allocating objects with sizes chosen from a fixed small set
  - e.g. {4KB, 8KB, …, 1MB}
- Finding best custom parameters can still be challenging
  - Needs experiments
- Use `pmempool` to sanity check the results
- Improper use of allocation classes results in very bad utilizations
  - E.g. allocate very small objects using our custom class 54…

Failures of Parallel Allocation

- Allocation failures can occur even though ample space remains in PMEM pool
  - Multi-threading
  - Variant object size
  - Customized allocation classes

Allocation Arena

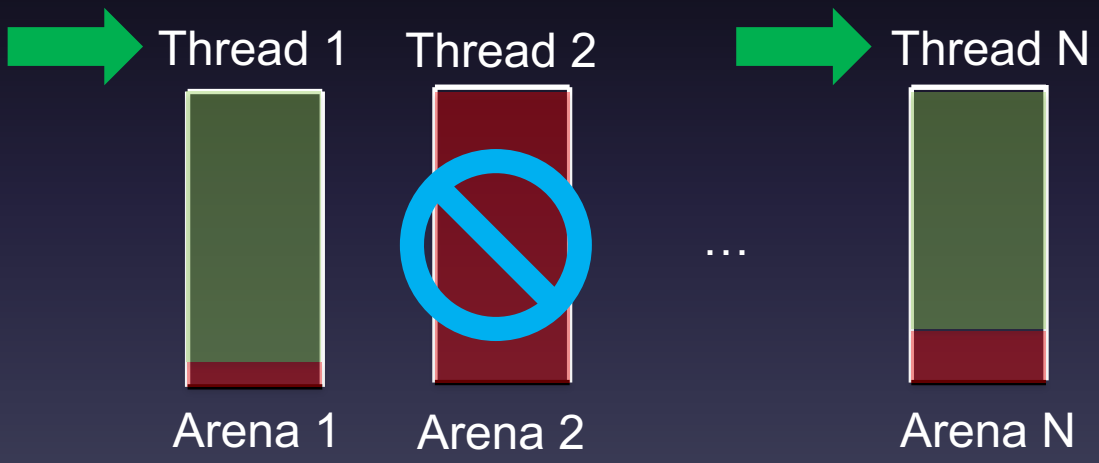
- `libpmemobj` uses arena to provide support for multithreaded allocations
  - Each thread is assigned a separate arena
  - High concurrency



Picture source: Intel PMDK webinar

A Hypothesis

- Unbalanced use of arenas
  - Arena reserve memory per allocation class
  - Thread is assigned to an arena at first allocation



To Validate Our Hypothesis

- Two sets of threads
  - Set 1: allocate objects in parallel

```
int alloc_objects(PMEMobjpool *pool, PMEMoid *objects,
                  unsigned object_count, size_t size, unsigned alloc_class_id) {
  for (unsigned i = 0; i < object_count; i++) {
    int rc = pmemobj_xalloc(pool, &objects[i], size, 0,
                            POBJ_CLASS_ID(alloc_class_id),
                            nullptr, nullptr);
    if (rc) {
      LOGERR("Allocate object %u size %u failed: %d",
             i, size, errno);
      return -errno;
    }
  }
  return 0;
}
```

May fail due to unbalanced allocation

  - Set 2: free objects allocated by Set 1 in parallel

```
int free_objects(PMEMobjpool *pool, PMEMoid *objects, unsigned object_count) {
  unsigned i = 0;
  int rc = 0;
  TX_BEGIN(pool) {
    for (; i < object_count; i++) {
      if (OID_IS_NULL(objects[i])) {
        continue;
      }
      pmemobj_tx_free(objects[i]);
    }
  } TX_ONABORT {
    rc = -pmemobj_tx_errno();
    LOGERR("Free object %u failed: %d", i, rc);
  } TX_END;
  return rc;
}
```

May also fail!

Transactional deallocation also requires allocating log in PMEM!

Solutions in PMDK 1.5

- Retry in user application via a different thread
- Arena pre-reservation

Solutions in PMDK 1.6

- PMDK 1.6 allows a thread to manually change arena
  - `pmemobj_ctl_get("heap.arena.[arena_id].size")`
  - `pmemobj_ctl_set("heap.thread.arena_id")`

- Multiple solutions can be developed based on this facility

Solutions

- Use single arena
  - Easy to implement
  - Hurt concurrency due to single lock contention
- Allocation class based arena
  - Simple to implement
  - Minor conflicts on arena (when you have many threads allocating same sized objects)

Solutions cont.

- Choose a different arena upon allocation failure

Evaluation

- Test bed
  - CPU: Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz 96 cores
  - Dax: 1 AEP Dimm 128G
- Methodology
  - 4 threads allocate objects in parallel
  - 8 threads free objects in parallel
  - Object size: 4KB, 8KB, …, 64MB, one allocation class per object size
  - Stop after total allocated 4TB objects in total
  - Metric: the average time to stop the execution

Results

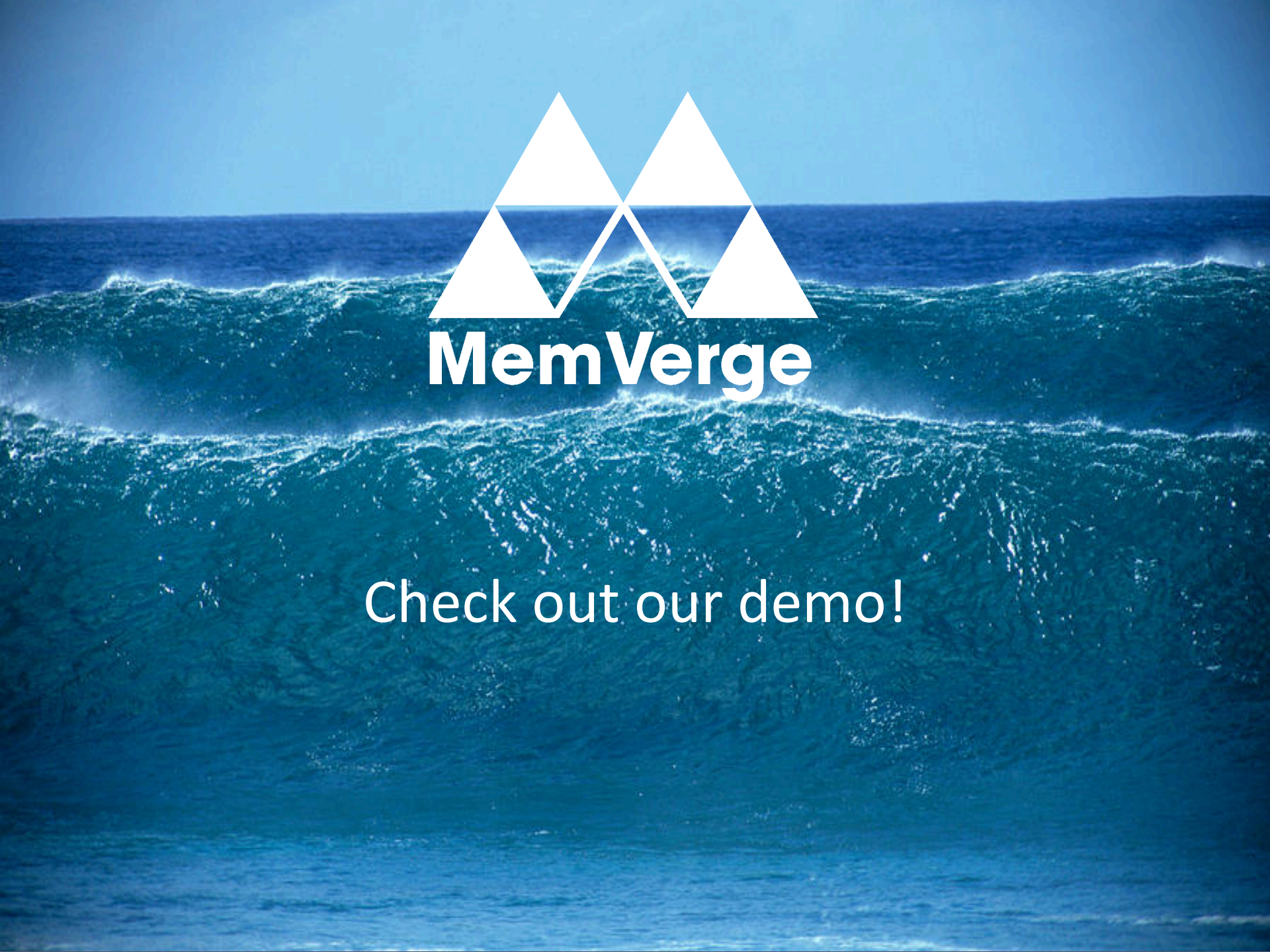
Solution	Time
Single arena	46.092s
Use arena for every allocation class	41.072s
Retry allocation in other arena	33.078s

* Allocations are fully randomized, results may change for other allocation patterns

Summary

- PMEM poses new challenges to software and hardware
  - High performance, high memory utilization, less fragmentation…
- SPDK, DPDK, PMDK are extremely useful as components in a distributed system.
- PMDK has ways to increase performance and productivity for PMEM applications
  - but requires advanced programming techniques and developer attention.
  - memory allocation is only a small tip of the iceberg
- MemVerge and Intel together deliver better scalability and performance at a lower cost via a disruptive PMEM-optimized data infrastructure
  - AI, Big Data, Banking, Animation Studios, Gaming Industry, IoT, etc.
  - Machine learning, Analytics, and Online systems

- We are in Beta! To become our PoC customer
  - https://www.memverge.com/#beta
- Join us! We are hiring strong engineering candidates
  - Locations: Shanghai, Beijing, Silicon Valley
  - https://www.memverge.com/careers/


MemVerge
Check out our demo!