POSEIDON: A Safe and Scalable Persistent Memory Allocator

Anthony Demeri, Wookhee Kim,
R. Madhava Krishnan, Mohammed Ismail,
Changwoo Min

VIRGINIA TECH™

Finally, NVM is here.

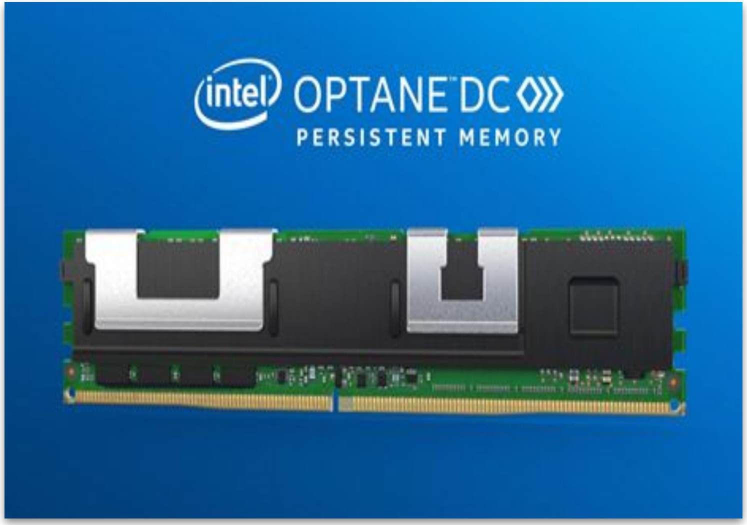
We are on the cusp of a **new era** for memory hierarchies.

Persistent memory has arrived!

Non-volatile Main Memory (NVM) was first commercialized by Intel as **OPTANE DC Persistent Memory**

It offers significant hierarchy benefits.

NVM = DRAM + Disk?



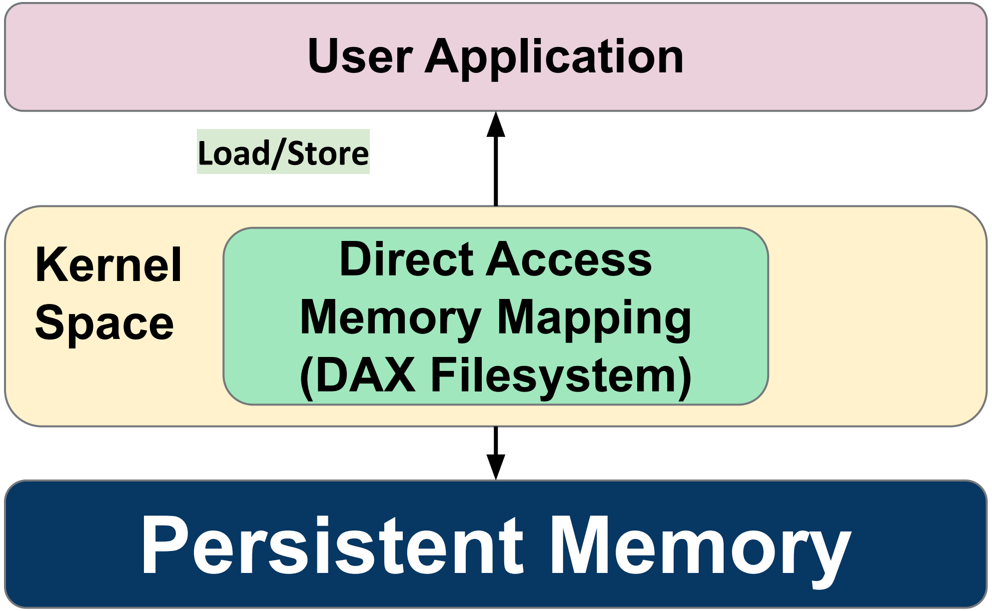
Non-Volatile Main Memory

Benefits of NVMM

- Read/write latency on order of DRAM
- **Byte-addressable** like the DRAM
  - Allows programmers to use common load/store instructions
- 100x faster than traditional disks
- Non-volatile like the storage
  - Retains data across failures

How does a developer access NVMM?

Direct Memory Mapping



NVM is accessed using DAX mode.

Entire NVM region is mapped to the user region.

But, who will now manage the large mmaped region?

A **NVM allocator** is needed.

Design Requirements-- NVM Allocators vs DRAM Allocators

Similarities

- Scalable to many-cores
- High -performance

Special Considerations

- Ensure crash consistency for the allocation/deallocation
- Crash consistent metadata for correct recovery
- Handle persistent memory leaks
- Safe (heap metadata protection)

Unfortunately, existing persistent memory allocators ***fall far short***

- Scalable only for small size memory allocations [Makalu-oopsla 16]
- Scales poorly for large size and number of allocations [Makalu-oopsla16]
- **No metadata safety [Makalu-oopsla16, PMDK]**
- Non-scalable across NUMA domain [Makalu-oopsla16, PMDK]

Persistent Memory Corruption in PMDK

1) Allocate all memory from the heap

2) Corrupt the header by "accidentally" writing prior to the returned pointer to feign a larger size

3) Erroneously free the larger size

4) Attempt to allocate again

5) PMDK over allocate already allocated memory

```
void pmdk_overlapping_allocation(nvm_heap *heap) {
    void *p[1024], *free;
    int i;

    /* Make the NVM heap full of 64-byte objects */
    for (i = 0; true; i = ++i % 1024) {
        if ( !(p[i] = nvm_malloc(64)) )
            break;
    }

    /* Free an arbitrary object but before freeing
     * the object, corrupt the size in its allocation
     * header to larger number. It will make the PMDK
     * allocator corrupt its allocation bitmap. */
    free = p[i/2];
    *(uint64_t *)(free - 16) = 1088; /* Corrupt header!!! */
    nvm_free(free);

    /* Since only one object is freed, the NVM heap
     * should be able to allocate only one 64-byte object.
     * But due to the allocation bitmap corruption
     * in the previous step, 9 objects will be allocated.
     * Unfortunately, 8 out of 9 will be already allocated
     * objects so it will cause user data corruption. */
    for (i = 0; true; i = ++i % 1024) {
        if ( !(p[i] = nvm_malloc(64)) )
            break;
        assert(p[i] == free); /* This will fail!!! */
    }
}
```

Summary of Problem

- Using NVMM ***requires more than a file system interface***, in order to capitalize upon low latency benefits

- Interfacing with NVMM directly is non-trivial; mandating the ***need for a persistent memory allocator***

- A persistent allocator ***must*** provide:

  (1) scalable performance to manycores

  (2) protection of heap metadata

Poseidon Presentation Outline

- Architectural overview
- Fundamentals overview
- Design decisions: compare and contrast
- Evaluations
- Conclusion

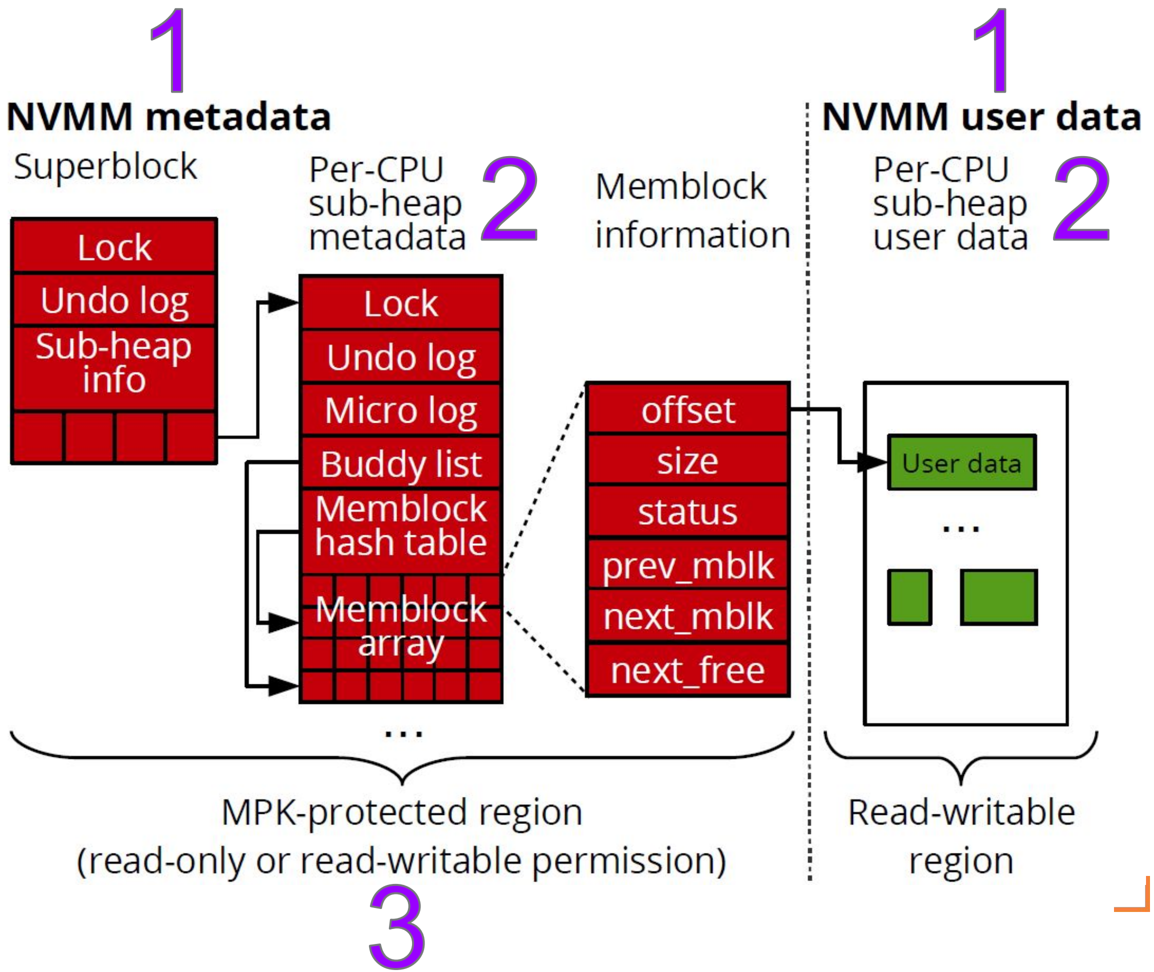
POSEIDON Architecture

Key Points

1 Metadata is stored **separately** from user data

2

3



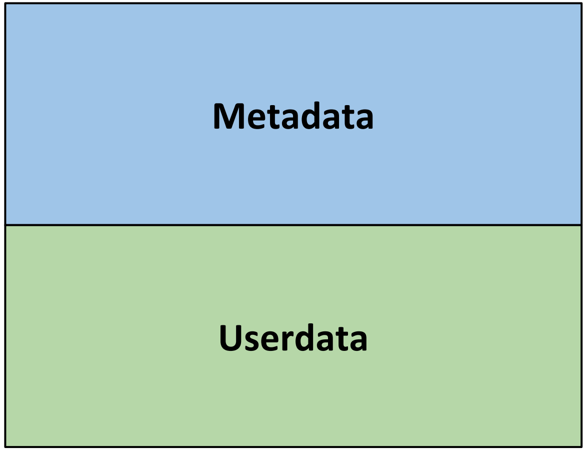
Poseidon Presentation Outline

- Architectural overview
- Design decisions: compare and contrast
- Evaluations
- Conclusion

How can we **protect metadata**?

Intel **Memory Protection Keys (MPK)**

Binning metadata storage is critical



Procedure

Initialization (one time cost)

```
int mpk_key = initialize_mpk_key(); // Get key
mpk_set_permissions(mpk_key, PROT_NONE); // Prot
mpk_map(addr, sizeof(meta), mpk_key); // Map meta
```

Updating

```
mpk_set_permissions(mpk_key, PROT_RDWR);
```

Completion

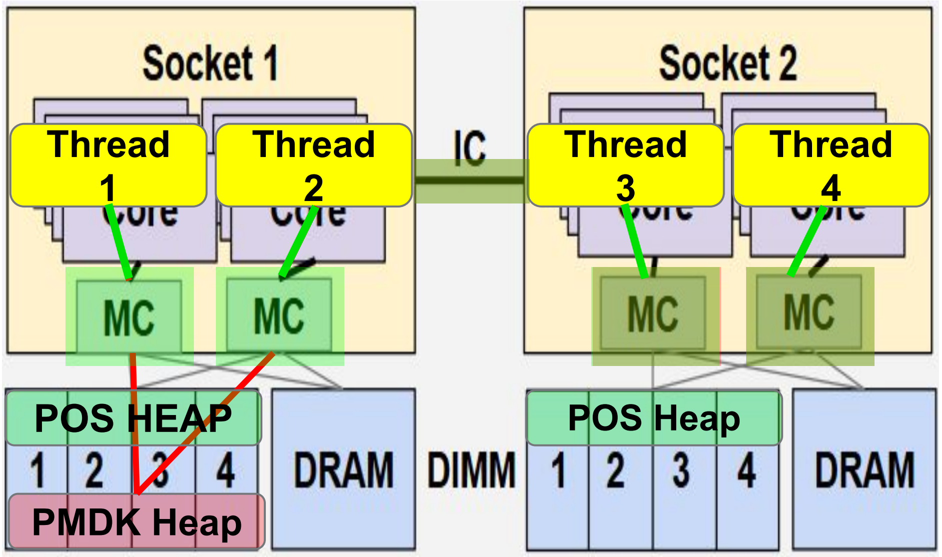
```
mpk_set_permissions(mpk_key, PROT_NONE);
```

Scalable

How can we **allow manycore scalability**?

Sub-heap, per-cpu design

- Existing allocators have been shown to introduce problematic bottlenecks via global lists
- We adopt per-CPU metadata structures
  - Minimizes inter-socket memory accesses
  - Maximizes use of memory controllers
  - Eliminate global system bottlenecks



More on paper..

- How poseidon handles API misuse?
- How Poseidon handles defragmentation?
- Compare and contrast Poseidon with the other memory allocators at the design level
- Implementation details

Poseidon Presentation Outline

- Architectural overview
- Fundamentals overview
- Design decisions: compare and contrast
- Evaluations
- Conclusion

How can we evaluate POSEIDON?

- **Discuss** the **metadata safety guarantee** of POSEIDON (refer paper)

- **Evaluate** the **scalability** of POSEIDON, relative to the other persistent memory allocators (PMDK and Makalu)

- **Evaluate** the performance of **real-world applications** with POSEIDON to demonstrate POSEIDON's impact

Benchmarks Overview-- For Scalability Evaluation

- Microbenchmarks
  - Raw allocator performance
- HPC benchmarks
  - Impact of memory location
- Real world benchmarks (refer paper)
  - YCSB (key-value store)
  - Larson (server simulation)

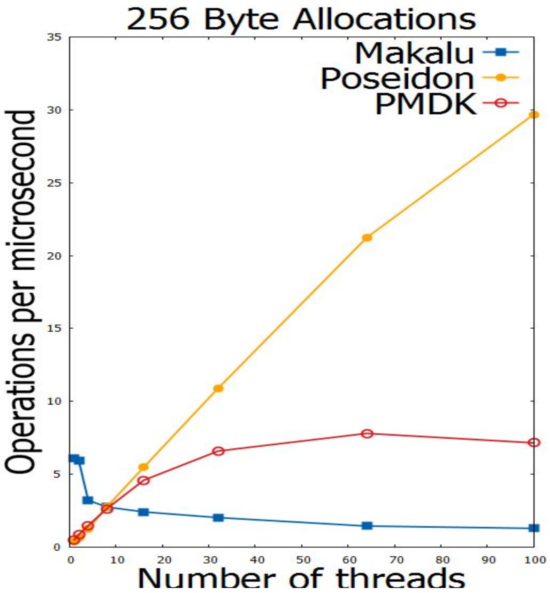
System Setup

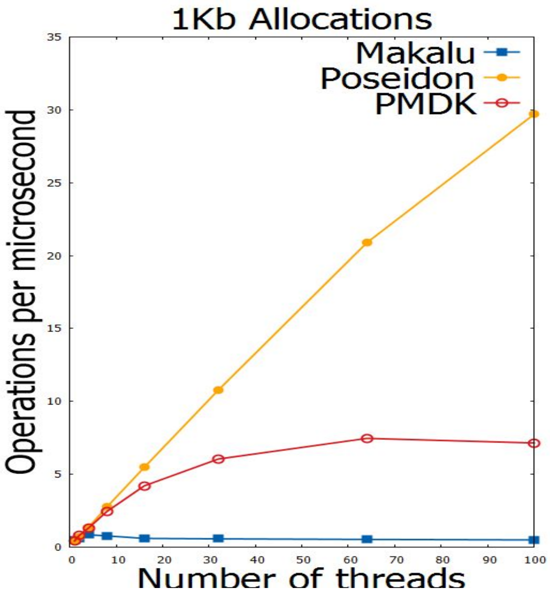
- Intel Xeon Platinum 8280M CPU (2.70 GHz)
- 768GB DRAM
- 3TB (12 x 256GB) Intel Optane DC Persistent Memory
- 2 Sockets
- 56 cores (112 logical cores)

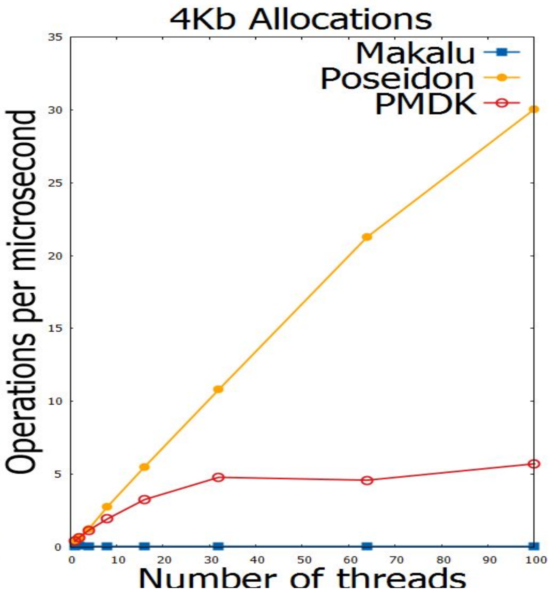
Microbenchmarks

Scalable
Performance

- Allocate 100 blocks, free 100 blocks in **random order**, repeat **x1,000,000**
- Not all memory allocators maintain free lists at the **per-CPU level**
- Makalu, for example, maintains a **global list** for allocations > 400 bytes







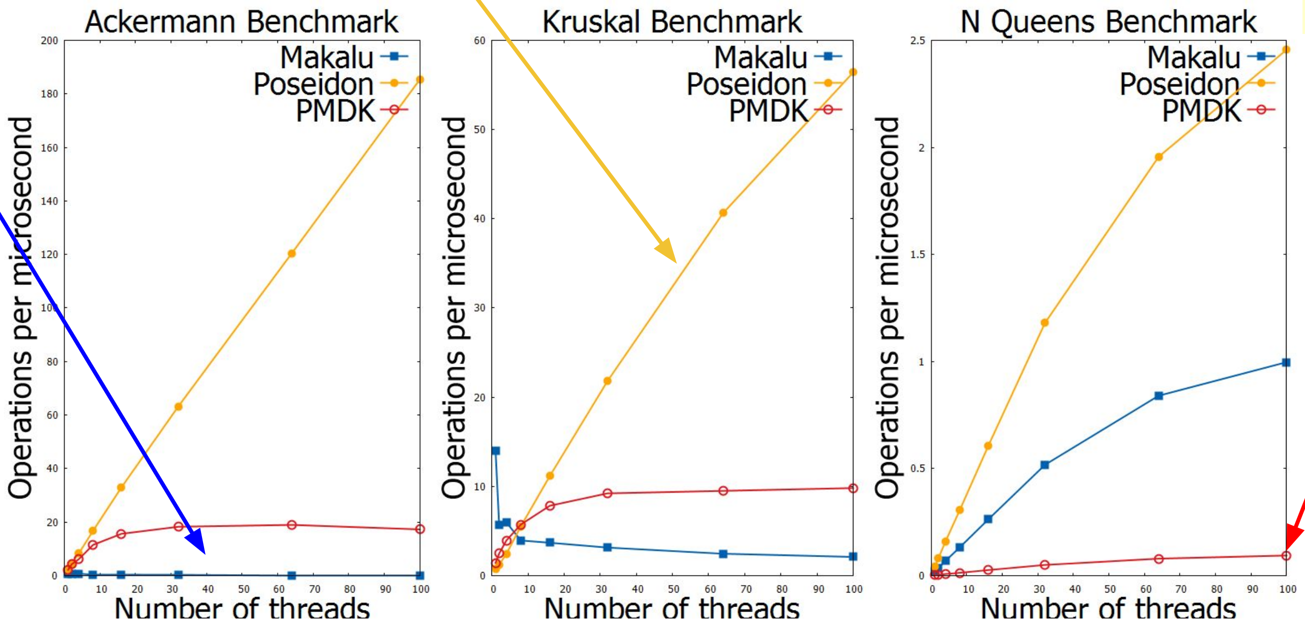
HPC Benchmarks

Scalable Performance

Makalu is not scalable for allocations > 400 bytes, due to its global free list

The per-CPU design and size-agnostic allocation design allows POSEIDON to scale linearly

The interconnect bottleneck of PMDK causes complete saturation of scalability



More Evaluations in the paper..

- Real-World Server Benchmark -- Larson Benchmark
- Integration and Evaluation with Index Structures -- YCSB Benchmark
- Proofs on how Poseidon achieves its safety guarantees

Conclusion

- We presented Poseidon- A safe and scalable persistent memory allocator
- Poseidon guarantees safety using Intel MPK
- Poseidon's’ per-cpu sub-heap design enables to scale almost linearly
- Consistently outperform its competitors while providing safety guarantees

Thank You

Safe

How can we **prevent API misuse?**

Multi-level Hash Table

- **API misuse can cause persistent metadata corruption**

- **If a memory allocator allows erroneous-frees or double-frees to corrupt allocator metadata, we have either persistent memory leaks or overallocation**

- **We must have a high performing, scalable means of verifying existing allocations**

Flowchart

User Frees Address "a"