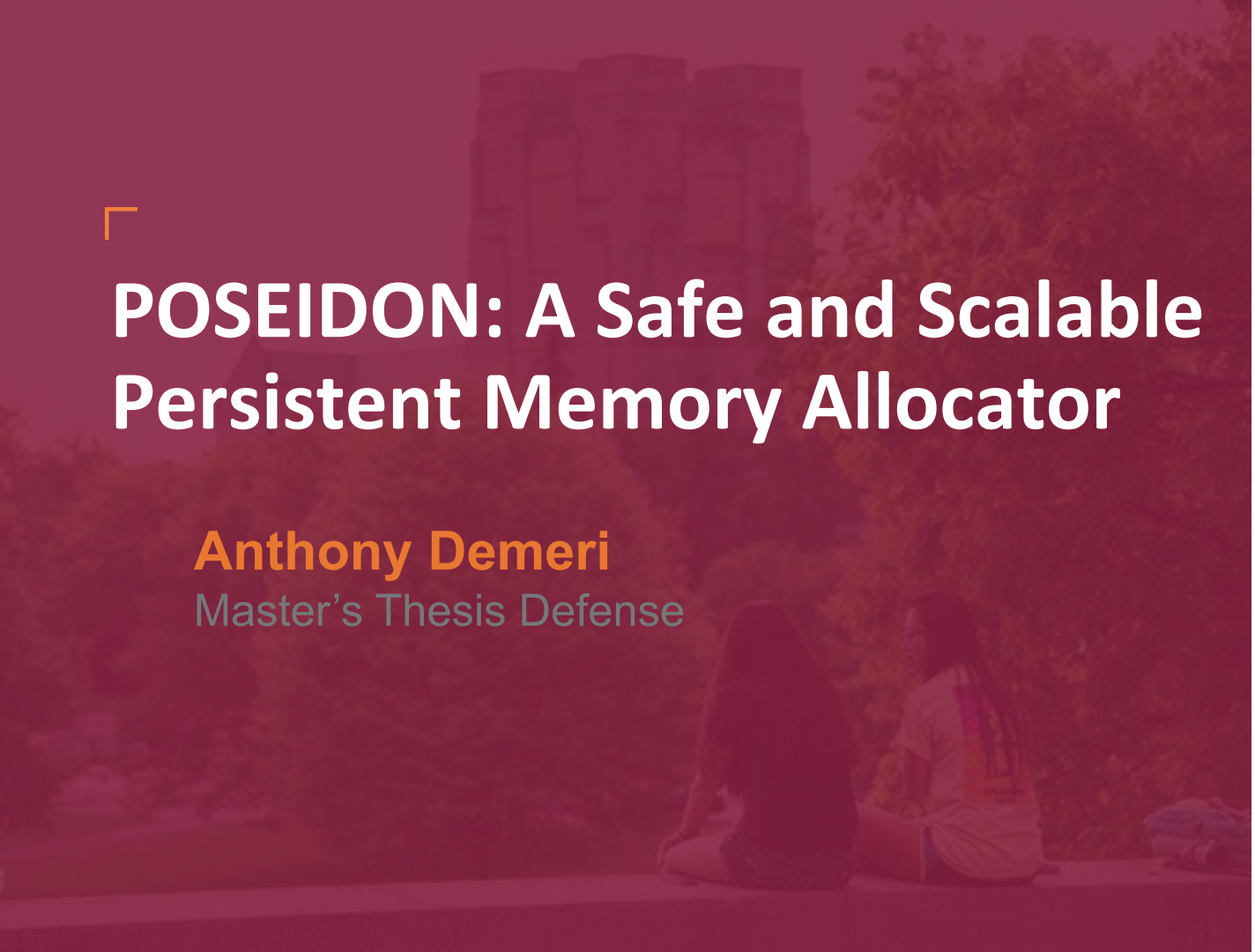


POSEIDON: A Safe and Scalable Persistent Memory Allocator

Anthony Demeri

Master's Thesis Defense

VIRGINIA TECH™

Finally, NVM is here.

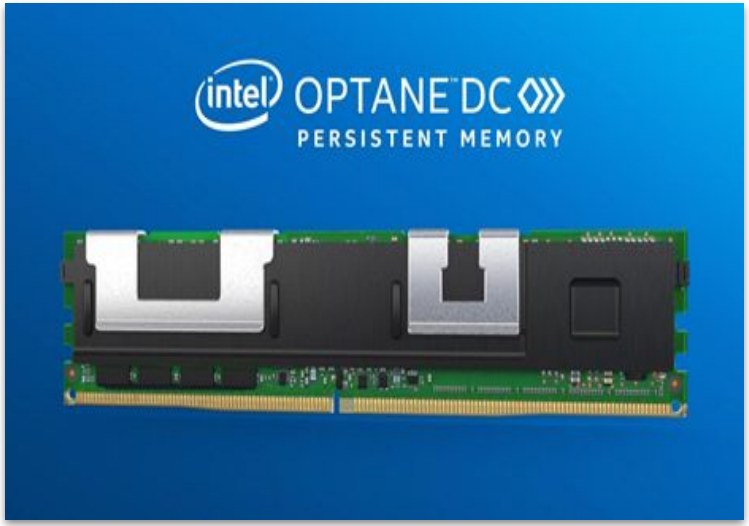
We are on the cusp of a **new era** for memory hierarchies.

Persistent memory has arrived!

Non-volatile Main Memory (NVMM) 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

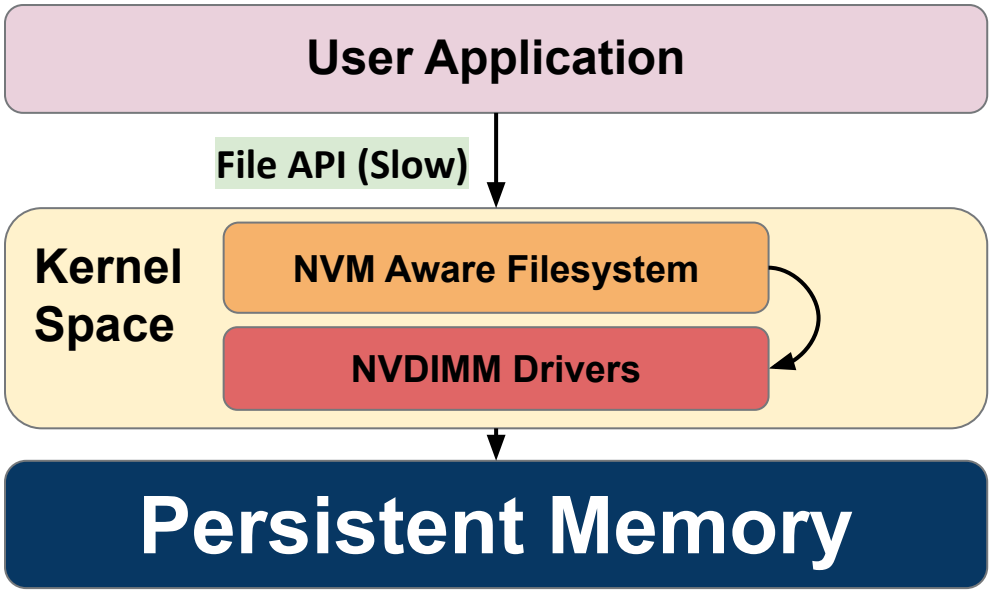
- Read/write latency on order of DRAM
  - 2-3x slower read
  - 4-8x slower write
- Greater memory density than DRAM
- Reduced energy consumption
- Low price-per-bit ($4/GB vs $35/GB)
- Non-volatile

Why does physical NVMM introduce challenges?

How does a developer access NVMM?

How does a developer access NVMM?

Case A. Using an **NVM Aware Filesystem**



Is this **acceptable?**

Pros:

- A legacy application can use NVMM without an API change

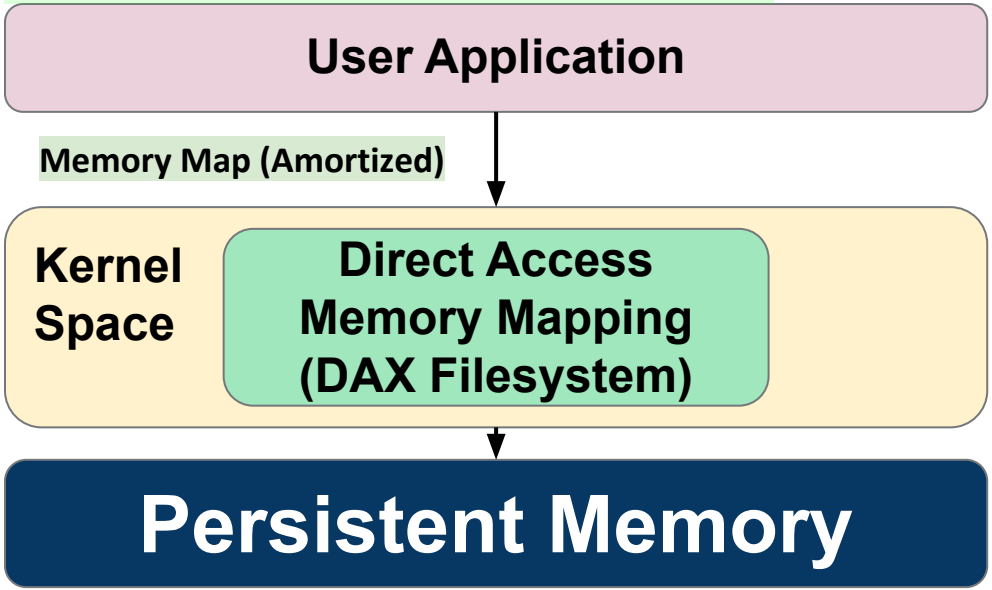
Cons:

- An NVM aware file system introduces traditional, significant overheads to prevent data corruption, which renders the latency benefits of NVMM **obsolete.**

How does a developer access NVMM?

How does a developer access NVMM?

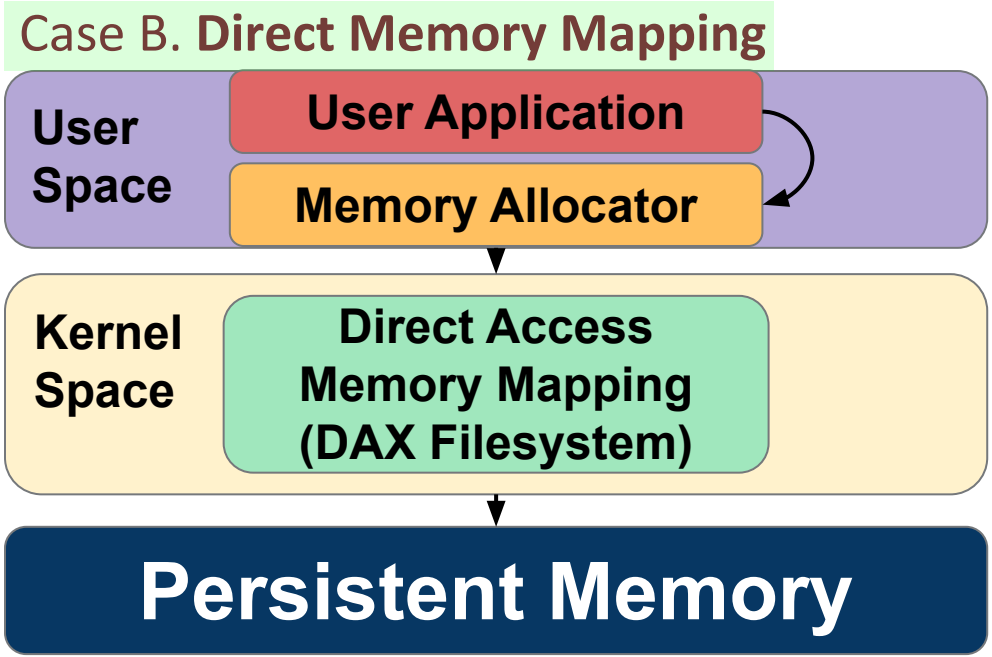
Case B. **Direct Memory Mapping**



Since the memory mapping is amortized and direct, we no longer have high overhead.

But, who will now manage the large mmaped region?

How does a developer access NVMM?



A ***memory allocator*** serves as the interface between an application and direct access (DAX) to NVMM, providing:

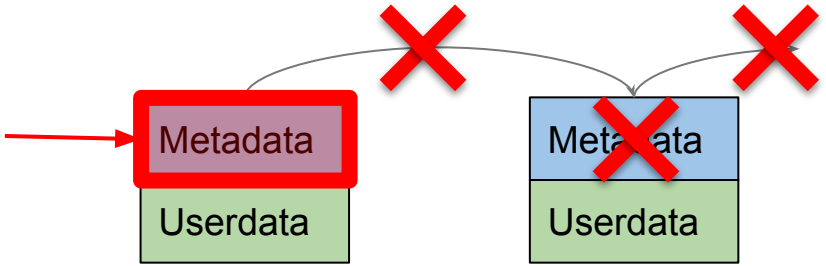
- Load/store access at the byte level
- Intuitive API (mirroring malloc/free)
- Abstraction of low-level details
- Often these mapped regions of contiguous memory are called **subheaps** or **pools**

Background: Memory Allocator Design

Metadata Fundamentals

- **Purpose**
  - Safely disseminate memory regions to processes
- **Required Storage**
  - Free addresses
  - Used addresses
- **Dangers**
  - Memory leakage
  - Corruption (mismatched free/used addresses)

Protection of metadata is ***critical***. Consider:




```
char buffer[16] = "i am corrupted\n";
```

```
char* addr = (char *) malloc(someBytes);
```

```
// We corrupted our metadata!
memcpy((addr - 8), buffer, sizeof(buffer));
```

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 overallocates 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!!! */
    }
}
```

Persistent Memory Leaks in PMDK

1) Allocate all memory from the heap

2) Corrupt the header by “accidentally” writing prior to the returned pointer to feign a smaller size

3) Erroneously free smaller sizes

4) Attempt to allocate again

5) PMDK fails to allocate the same size; the memory has been *persistently* leaked

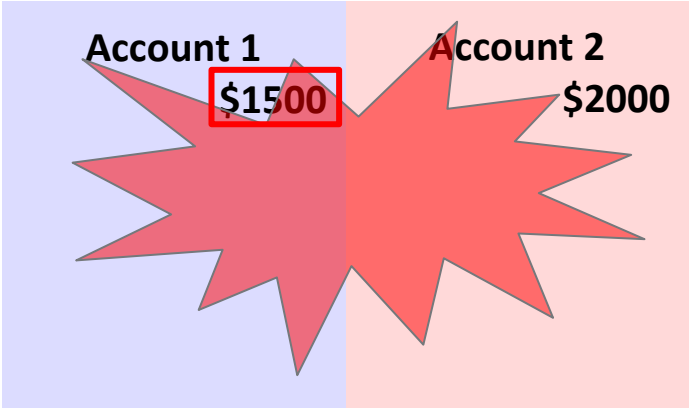
```
void pmdk_permanent_leak(nvm_heap *heap) {
    static void *p[INT_MAX];
    int i, nalloc;

    /* Make the NVM heap full of 2MB objects */
    for (i = nalloc = 0; true; i++, nalloc++) {
        if ( !(p[i] = nvm_malloc(2*1024*1024)) )
            break;
    }

    /* Free all allocated objects but before freeing objects,
     * corrupt the size in their allocation header to smaller
     * number. It will make the PMDK allocator corrupt chunk
     * headers to smaller in size. */
    for (i = 0; i < nalloc; i++) {
            *(uint64_t *)(p[i] - 16) = 64; /* Corrupt header!!! */
      nvm_free(p[i]);
    }

    /* Since all objects were freed, the NVM heap should be
     * able to allocate the same number of 2MB objects. But
     * due to the chunk header corruption in the previous
     * step, there is no such free chunk larger than 2MB.
     * Thus, allocation will fail. */
    for (i = 0; true; i++) {
        if ( !(p[i] = nvm_malloc(2*1024*1024)) )
            break;
    }
    assert(i == nalloc); /* This will fail!!! */
}
```

NVMM is **fundamentally different** than DRAM

Special Considerations

- Protection of heap metadata persistent data corruption
  - ACID Principles
    - Atomicity
    - Consistency
    - Isolation
    - Durability
- Persistent memory allocator
  - Safe (heap metadata protection)
  - Scalable to manycores
  - High performance

We ***must*** have crash consistency!



Let the owner of "Account 2" **owe** the owner of "Account 1" $500

Is guaranteeing ***crash consistency*** the only challenge?

Unfortunately, it is only ***one of many challenges*** for a persistent memory allocator.

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

Pallocator [VLDB'17]

Intel's PMDK (the defacto standard)

Makalu [OOPSLA'16]

PMDK Architecture

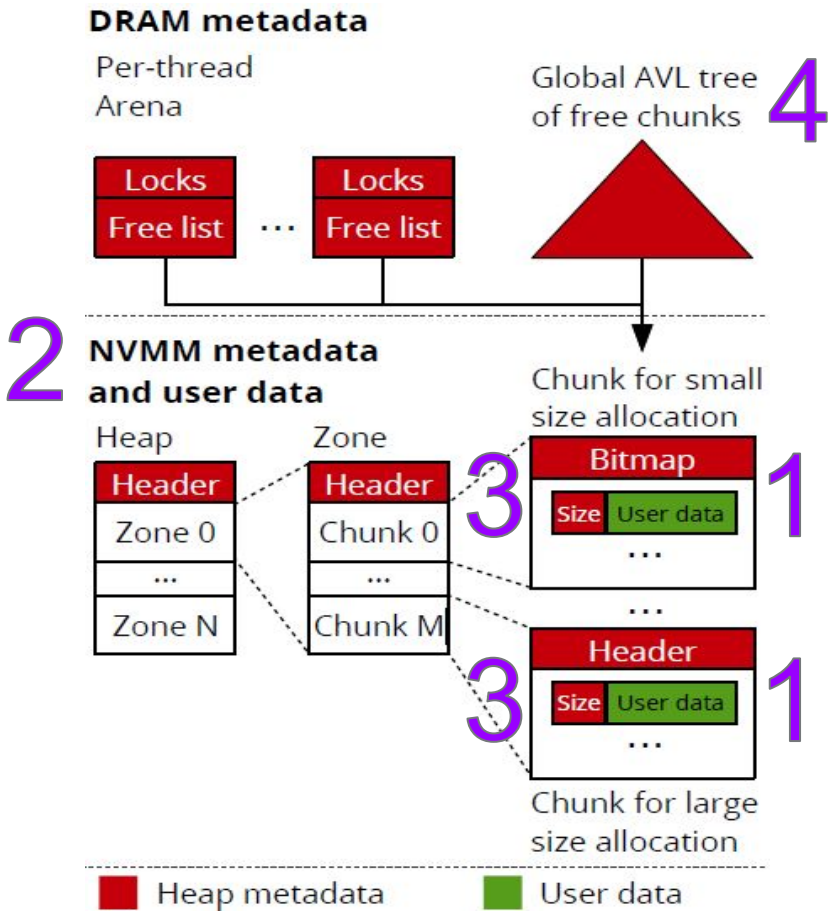
Keypoints

1 Metadata is stored **in-place**, with user data

2

3

4



Design Issues with PMDK

Design Decision	Goal	Impact
In-place metadata	Traversal of metadata	**Metadata is unprotected**
Global AVL Tree	Distribution of free chunks	**Scalability bottleneck**
Ignoring metadata protection	Optimize performance	**DANGER! Persistent corruption**
Coupling concurrency with heap architecture	Minimize contention	**Introduce performance Bottleneck via socket interconnect**

Summary of Problem

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

Thesis Goals

- ***Investigate*** an appropriate design using the amalgamation of known and novel techniques from various technical works and research

Poseidon Presentation Outline

- Architectural overview

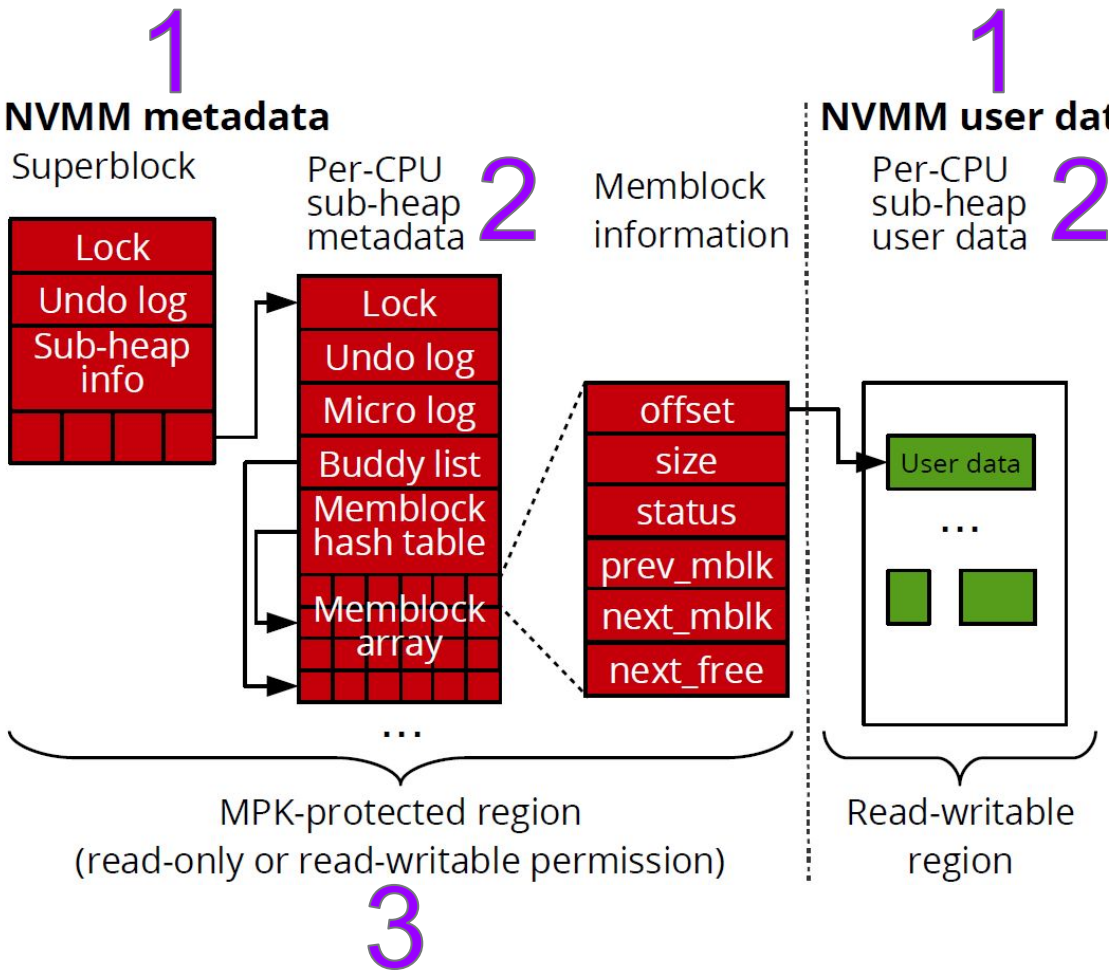
POSEIDON Architecture

Keypoints

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

2

3



Poseidon Presentation Outline

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

POSEIDON was designed to be: ***safe, scalable, and high performance***

Safe

- **Protection** of heap metadata
- Adherence to **ACID** principles
- **Recovery** from system or application failure

Scalable

- Performance **scales linearly** with core increase
- Allows distribution of **sub-heaps**

High Performance

- Safety and scalability design choices **minimally impact critical path**

Technical Design Questions

How can we ***protect metadata***?

How can we ***recover from failure***?

How can we ***prevent API misuse***?

How can we ***grant atomic transactions***?

Safe

How can we support ***manycore scalability***?

Scalable

How can we ***mitigate memory fragmentation***?

How can we ***reduce metadata overhead***?

Performance

Poseidon Presentation Outline

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

VIRGINIA TECH

Safe

How can we **protect metadata**?

Safe

Memory Protection Keys

MPK (hardware primitive, available on all systems supporting NVMM)

- Requires a ***single*** mapping of a memory region (high latency is amortized)
- Modifications to access require only writing to a register
- The register is, by definition, a per-cpu/per-thread entity
- Multiple threads can thus have different memory permissions
- Allows scalable protection
- **Critical**: MPK provides protection with a minimum of ***page-size precision;*** *this is* ***not an option for PMDK****!*

Safe

How can we **prevent API misuse?**

Scalable

How can we **allow manycore scalability**?

Performance

How can we **reduce** metadata overhead?

Poseidon Presentation Outline

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

Implementation

- **POSEIDON Software**
  - 16,000 lines of **C code**

How can we evaluate POSEIDON?

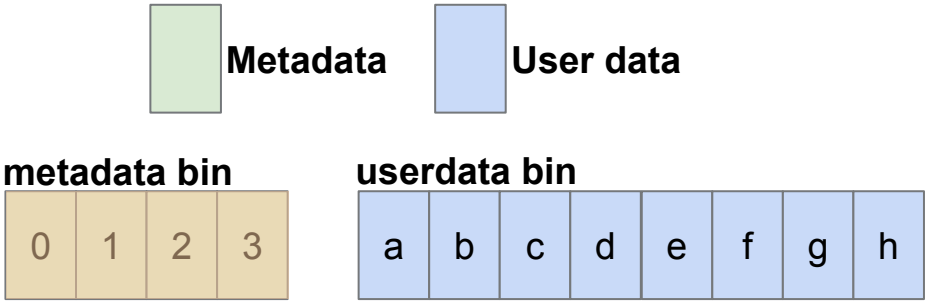
- **Discuss** the **metadata safety guarantee** of POSEIDON

Safe

Safety Guarantee

By using **MPK**, POSEIDON guarantees protection of heap metadata

- When a sub-heap is initialized, the metadata **given neither read nor write access**
- When POSEIDON needs to modify metadata, internally, it **enables read/write access** to **only** the executing processor (MPK registers)
- Any attempted writes from the user application will cause a SIGSEGV (segmentation fault)



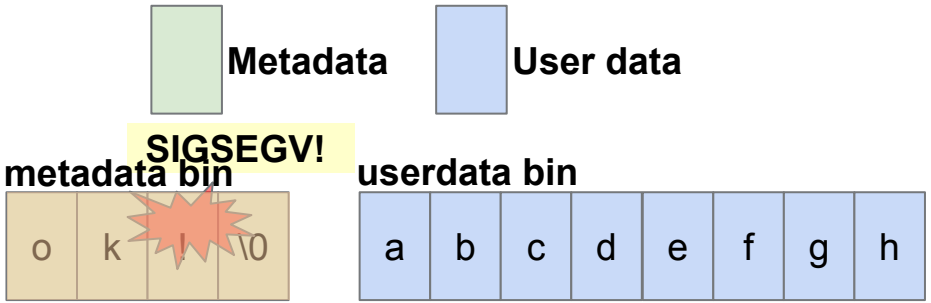

```
POS_WR(meta)
{
        ENABLE_RDWR()
        M[meta] = "ok!"
        DISABLE_RDWR()
}
```

Safety Guarantee

Safe

By using **MPK**, POSEIDON guarantees protection of heap metadata

- When a sub-heap is initialized, the metadata **given neither read nor write access**
- When POSEIDON needs to modify metadata, internally, it **enables read/write access** to **only** the executing processor (MPK registers)
- Any attempted writes from the user application will cause a SIGSEGV (segmentation fault)




```
POS_WR(meta)
{
        ENABLE_RDWR()
        M[meta] = "ok!"
        DISABLE_RDWR()
}
```

Benchmarks Overview

- **Microbenchmarks**
  - **Raw allocator performance**

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)

Competing Allocators Overview

Makalu

- **Does not protect** heap metadata
- **In-place** metadata
- **Global** list of free chunks
- Provides **no transactional allocation** capability
- **No explicit defragmentation**

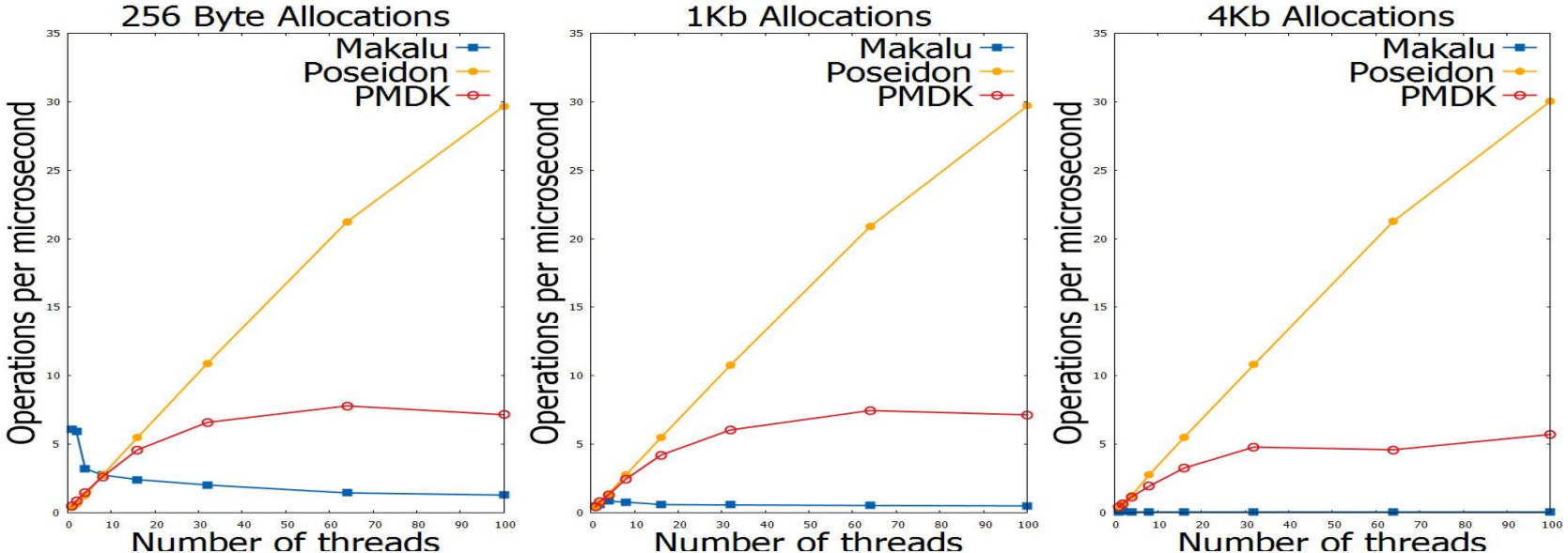
PMDK

- **Does not protect** heap metadata
- **In-place** metadata
  - Bitmap
  - Header
- **Global** AVL list
- **Coupled concurrency protection** with subheap
  - Per-thread arena

Scalable Performance

Microbenchmarks

- 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



High Performance Benchmarks

- **Ackermann (4, 5) x 100,000**
  - Large memory mapping (16Mb)
  - Persistent caching example
- **Kruskal (5th order) x 100,000**
  - 512B allocation size
  - Heavy memory read/write
- **N Queens (8x8) x 100,000**
  - 32B allocation size
  - Heavy memory read/write

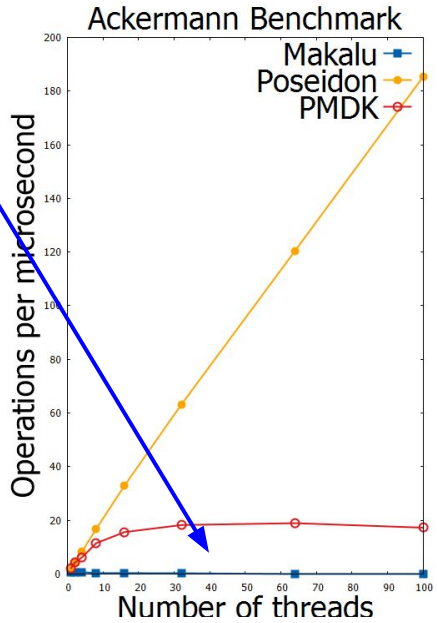
High Performance Benchmarks

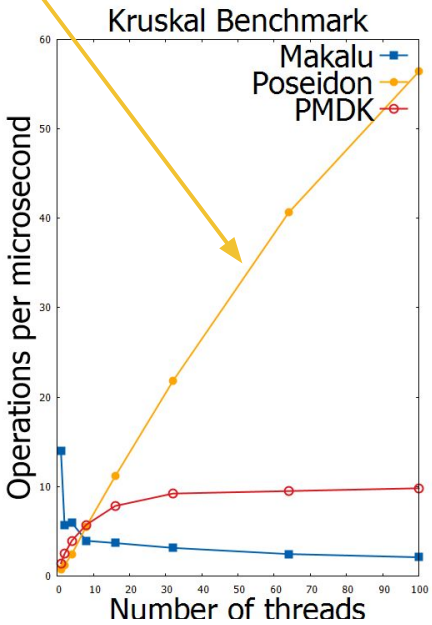
Scalable Performance

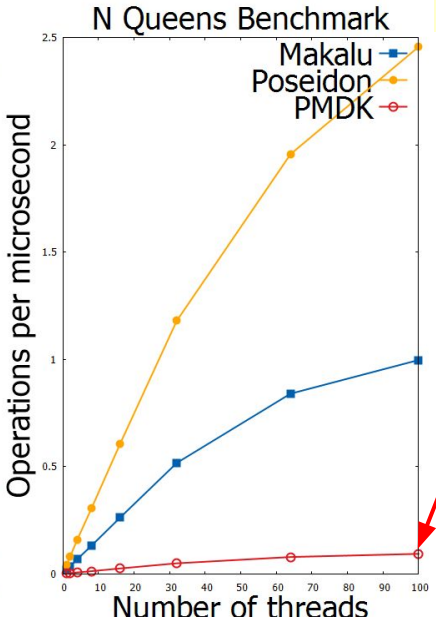
Makalu is not scalable for allocations > 400 bytes, due to its global free list

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

The interconnect bottleneck of PMDK causes complete saturation of scalability





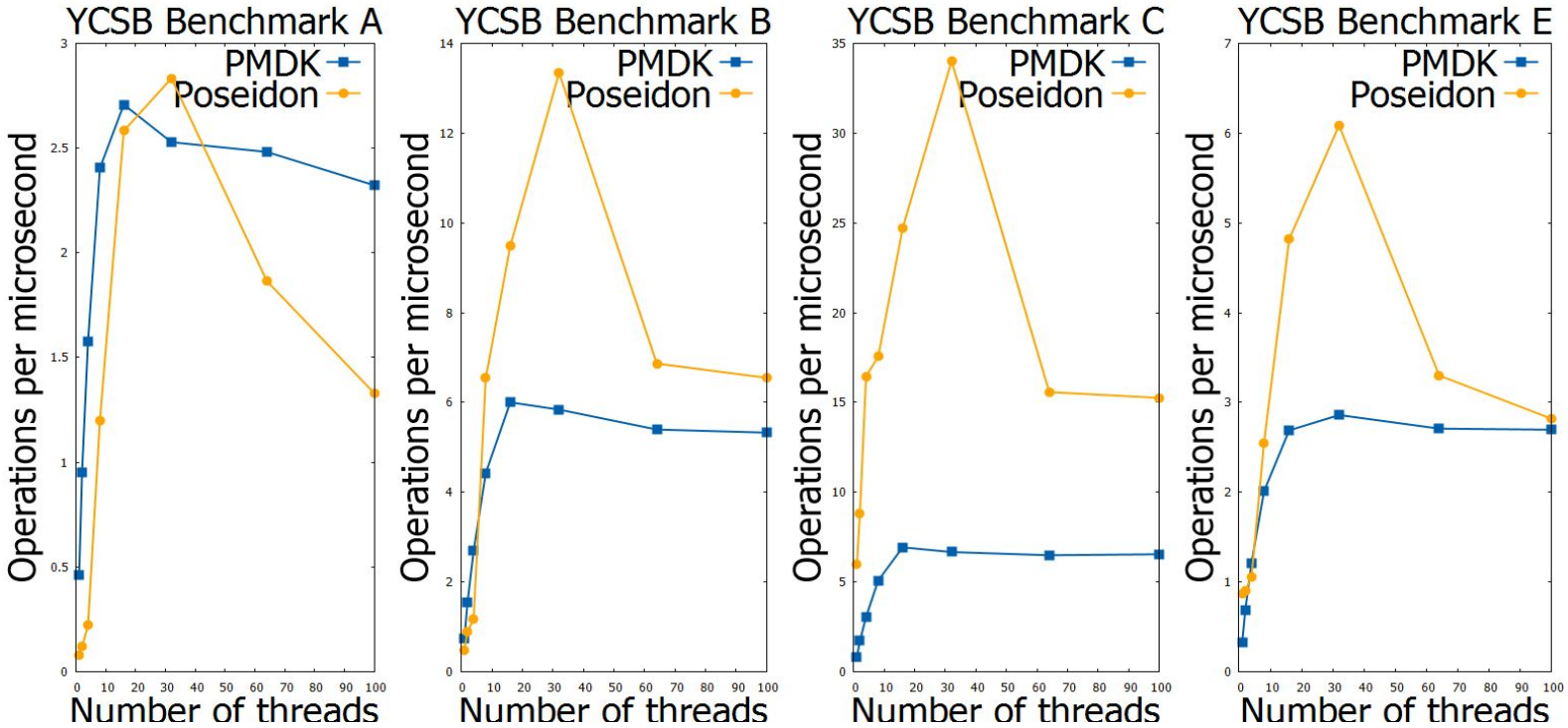


YCSB Benchmarks

- **Key-value Store with ART (Radix Tree) persistent data structure**
  - Request a shared heap from the main thread (danger! application bottleneck, but it is the standard)
  - Using multiple threads, allocate memory from the shared heap
  - Write, read, and scan memory regions from these threads
  - Free the memory from these threads

YCSB Benchmarks

Scalable Performance



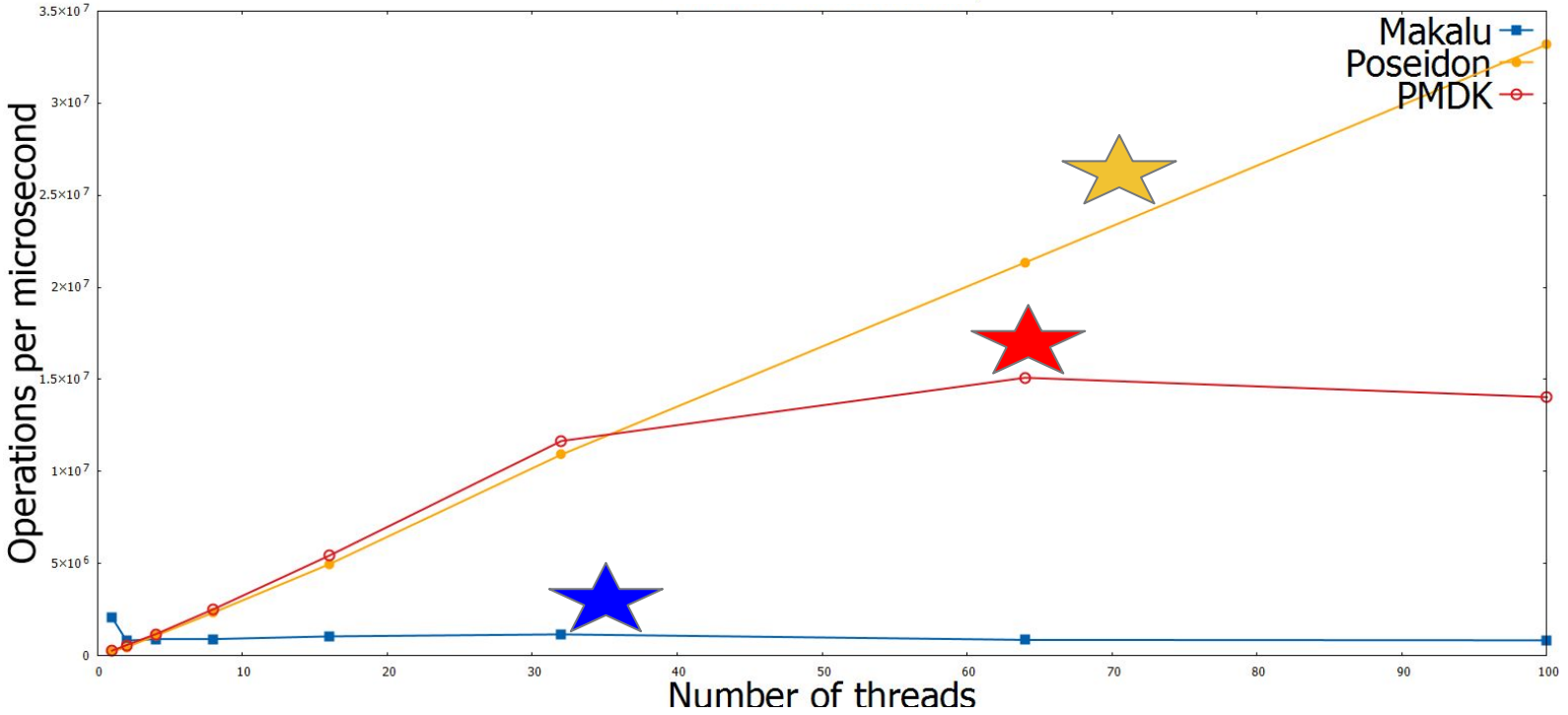
Larson Benchmark

Scalable Performance

- **Real World Server Simulator**
  - Map a universally accessible region of memory
  - “Warm-up” the region of memory with 512B allocations
  - Using multiple threads, access, update, and free arbitrary memory regions
  - Represents concurrent access of shared data and memory regions of dynamic values and sizes
  - A ***standard benchmark*** in academia for evaluating memory allocators

Larson Benchmark

Scalable Performance



Poseidon Presentation Outline

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

Conclusion

Technical Contributions

- A **safe, scalable, high performance** persistent memory allocator **does not yet exist**; PMDK's design decisions are such that safety and scalability vary inversely

POSEIDON: A Safe and Scalable Persistent Memory Allocator

Together, with POSEIDON, we can usher in a new era of memory hierarchy and performance gains

Anthony Demeri
demeri@vt.edu

Thank you!

VIRGINIA TECH

POSEIDON: A Safe and Scalable Persistent Memory Allocator

Together, with POSEIDON, we can usher in a new era of memory hierarchy and performance gains

Anthony Demeri
demeri@vt.edu

Thank you!

For reference, the following slides include additional background information; they are intentionally not included in the presentation due to time constraints

Technical Design Questions

POSEIDON Technical Contributions

Technical Questions	POSEIDON Solution
How can we ***protect metadata***?	Intel's ***Memory Protection Keys (MPK)***
How can we ***recover from failure***?	***UndoLogs*** *and* ***Startup Synchronization***
How can we support ***manycore scalability***?	***Subheap-local*** entities
How can we ***mitigate memory fragmentation***?	*Amortized* ***Iterative Defragmentation***
How can we ***reduce metadata overhead***?	*Memory* ***Hole-Punching***
How can we ***prevent API misuse***?	*Multi-level* ***Hash Tables***
How can we ***grant atomic transactions***?	Internal ***Micro-logging***

Background

General Memory Allocator Fundamentals

Practical Importance

- Software development requires arbitrary dynamic memory access
- Illogical to place this burden on an application developer

Design Decisions

- Free list storage
- Performance & scalability
- Metadata placement
- Metadata overhead
- Metadata protection
- Defragmentation
- Robustness (system or application failure)

Background

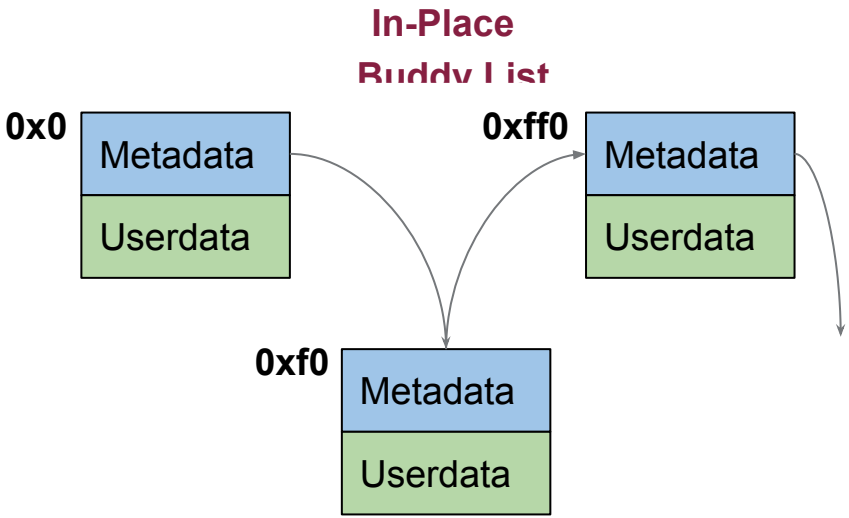
Memory Allocator Design

Memory Data Structures

- Slabs
- Buddy list

Metadata Data Structures

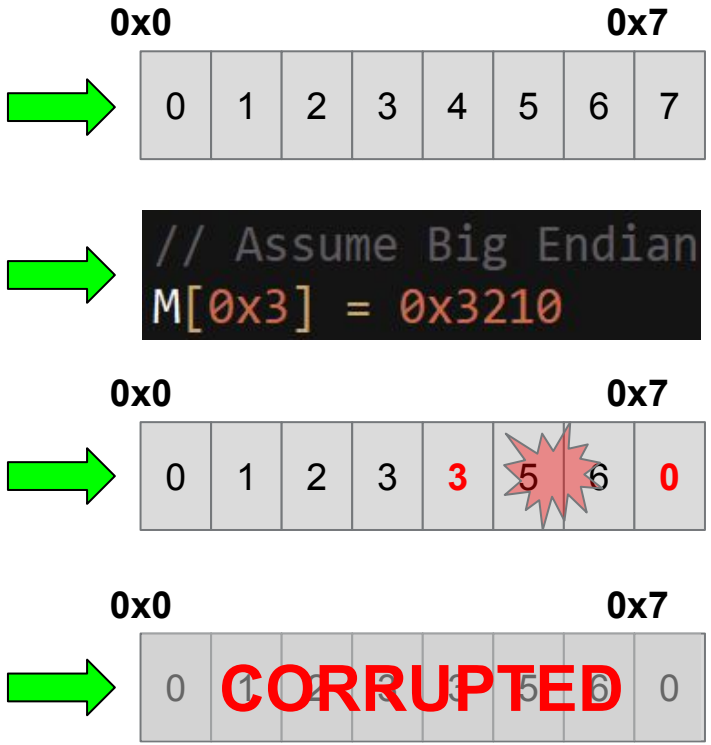
- In-place



Background

Atomicity

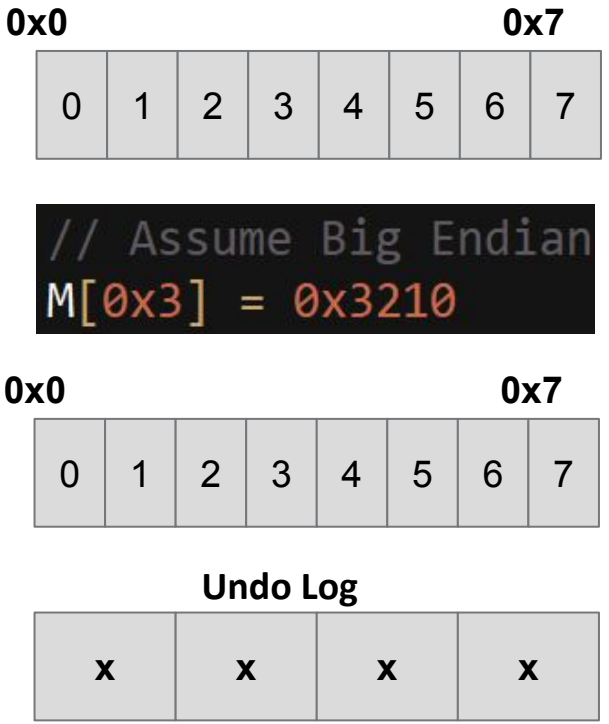
- Modifying non-volatile memory requires Specific attention to guaranteeing atomic Updates of data
- If an update is occurring and an application or system crash occurs, the state must not be corrupted.
- This means, any logical update must be atomic
- We can accomplish this by using Undo Logging



Background

Atomicity

- Modifying non-volatile memory requires specific attention to guaranteeing **atomic updates** of data
- If an update is occurring and an application or system crash occurs, the **state must not be corrupted**
- This means, any **logical update must be atomic**
- We can accomplish this by using **Undo Logging**



Background

Atomicity (transactional)

- **A developer must also have the ability to perform atomic transactions**

What happens here?

```
atomic_tx(void **a, void **b, void **c)
{
        (*a) = nv_alloc(size);

        (*b) = nv_alloc(size);

        (*c) = nv_alloc(size);
}

int main()
{
      void* a = NULL;
      void* b = NULL;
      void* c = NULL;

      while(1)
      {
            atomix_tx(&a, &b, &c);
            doWork(a, b, c);
      }
}
```

Background

Consistency

- A memory allocator must not interfere with the consistency semantics of a given application

```
update(int **a, int num)
{
      (*a) = nv_alloc(sizeof(int));

      (*(*a)) = 0x1 + num;
}

int main()
{
      int* a = NULL;
      int* b = NULL;

      update(&a, 0);
      // a == 1

      update(&b, *a);
      // b == 2
}
```

sfence();

Background

Isolation

- A persistent memory allocator must protect metadata from concurrent access

```
threadCallback(int** num)
{
        lock();
        (*num) = nv_alloc(sizeof(int));
}
        release();

int main()
{
        int* a = NULL;
        int* b = NULL;

        // Get memory in two separate
        // threads
        runThread1(&a);
        runThread2(&b);

        assert(a != b);
}
```

Background

Durability

- Updates made to the metadata must persist across system failure
- Any update to non-volatile memory, moreover, must survive a power-loss
- This can be accomplished by using the clwb() instruction, which flushes a cache line to the memory controller

```
update(int* addr, int val)
{
        sfence();
      (*addr) = val;
        clwb(addr);
}

int main()
{
      int  a = -1;
      int  b = 0;

      update(&a, b);
}
```

Design

How can we **recover** from system or application failure?

- **ACID conformity tangentially grants recovery from system or application failure**
- **When writing to persistent memory, every update is first logged in an heap-local UndoLog**
- **On application start, the UndoLog is checked for existing data**
- **If the UndoLog is non-empty, it is replayed and the state is restored**

Pseudocode

```
main():
      // Initialize POSEIDON
      NV_INIT_ALLOCATOR()

      // Run user application
      doWork()

# POSEIDON internal function
NV_INIT_ALLOCATOR():
      for current_subheap in all_subheaps:
            # Check UndoLog for the given subheap
            if current_subheap.undoLogIsEmpty
                  current_subheap.replayUndoLog()
```

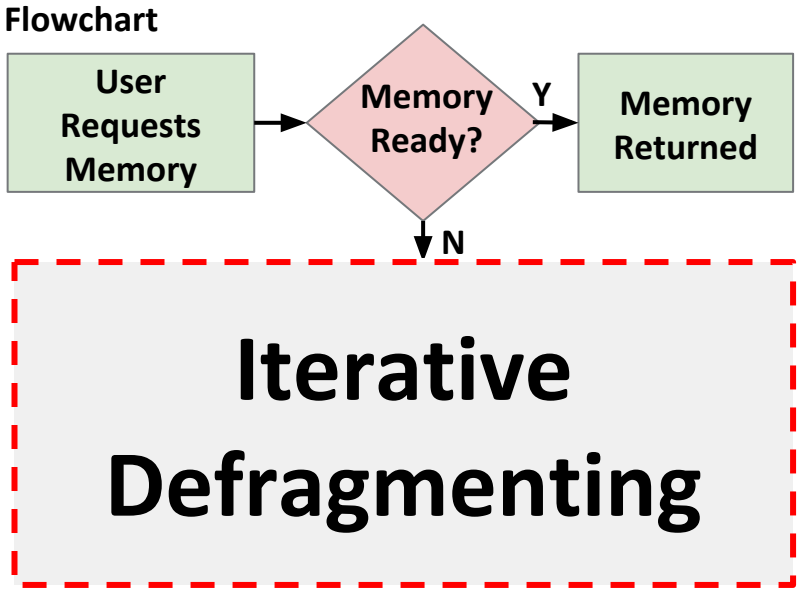
Undo Logging and start-up synchronization

Design

How can we **mitigate** memory fragmentation?

- Memory fragmentation is a prevalent issue for memory allocators
- We want to utilize maximum contiguous regions of memory
- We want to “keep the common case fast”
- POSEIDON’s defragmentation is iterative
  - Defragment the minimal number of block sizes
  - Iteratively defragment smaller block sizes on failure
  - Merge algorithm

Amortized Defragmentation



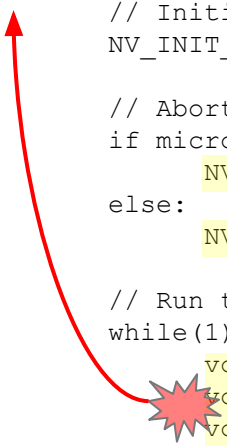
Design

How can we **grant atomic transactions?**

- **POSEIDON provides a micro-log, which holds internal transactional allocations**
- **When a developer has completed persistently storing their allocated memory regions, they *commit* their transactional atomically**
- **If a crash occurs during the transaction, the developer can either**
  - **(a) Abort the transaction and Free the memory**
  - **(b) Recover the transaction and continue**

Micro-logging

Pseudocode

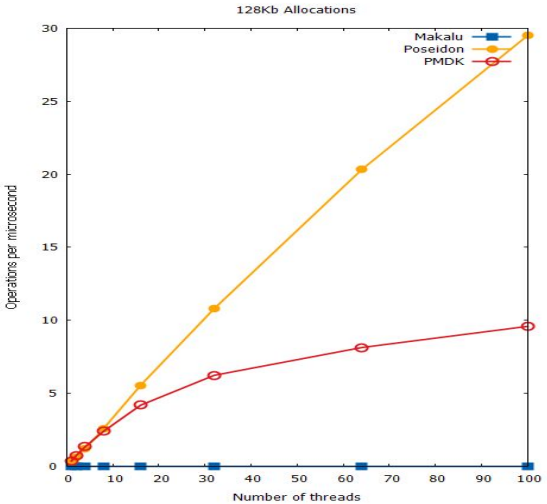


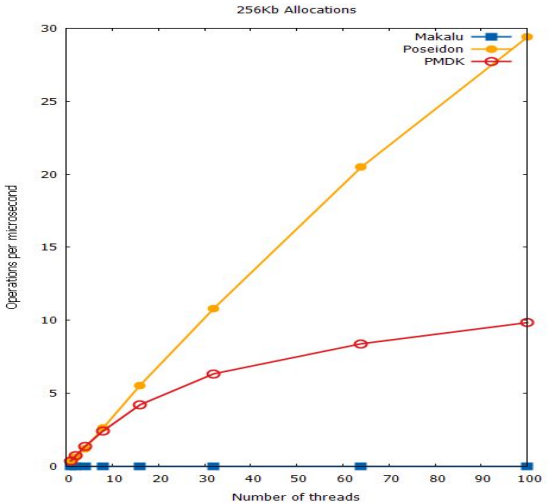
```
main()
        // Initialize POSEIDON
        NV_INIT_ALLOCATOR()

        // Abort any failed transactions
        if microLogNotEmpty() and lastTXIncomplete():
                NV_RELEASE_MALLOC_TX()
        else:
                NV_COMMIT_TX()

        // Run the user application
        while(1):
                void* a = NV_MALLOC_TX(size)
                void* b = NV_MALLOC_TX(size)
                void* c = NV_MALLOC_TX(size)

                storeUserPointers(a, b, c)

                NV_COMMIT_TX()
                doWork(a, b, c)
        return 0
```

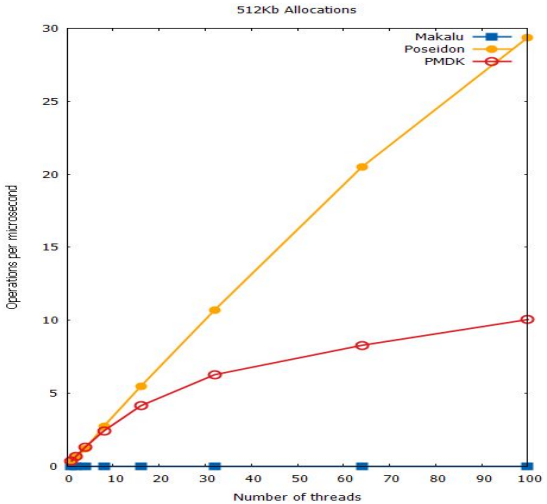
Evaluation

Microbenchmarks

- Memory allocators must **scale with allocation size**
- Unfortunately, Makalu's **large block list** becomes a major bottleneck



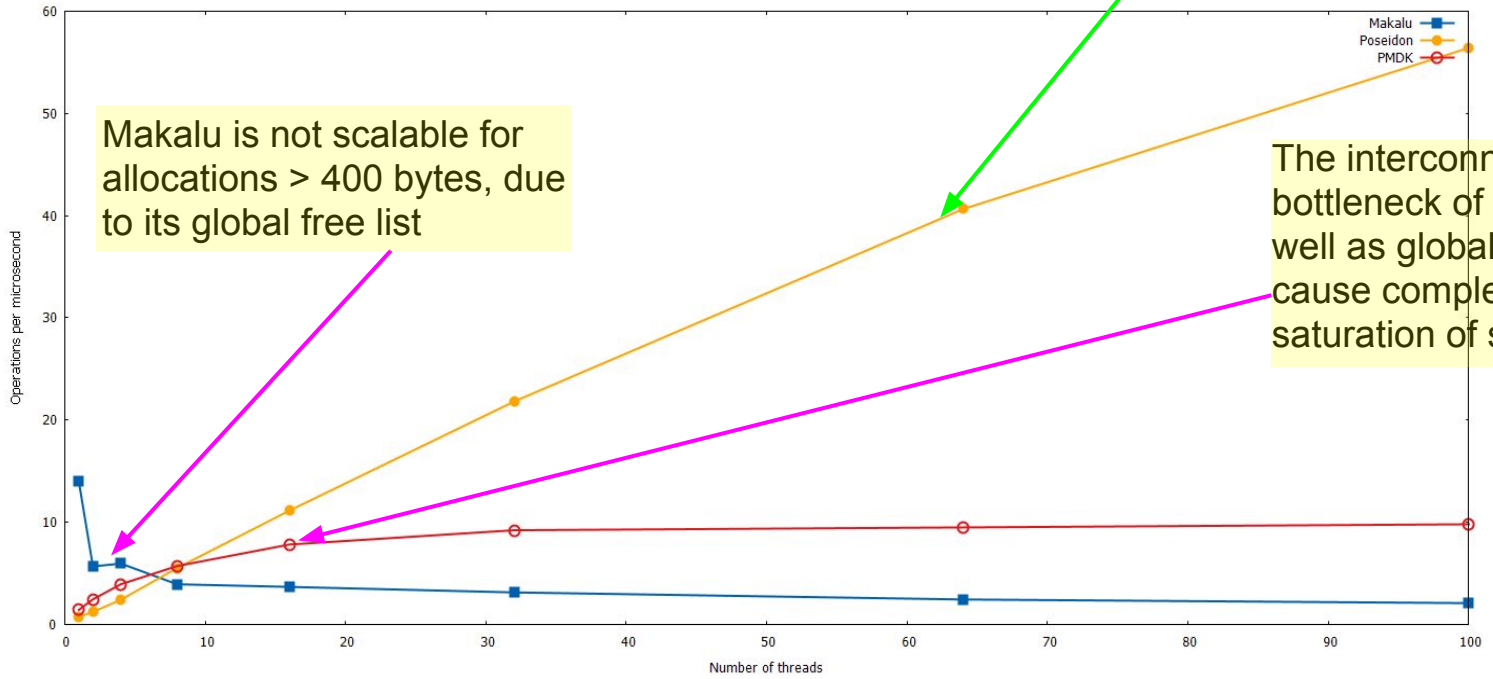




Evaluation

High Performance Benchmarks - Kruskal

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



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

The interconnect bottleneck of PMDK, as well as global AVL tree, cause complete saturation of scalability

Evaluation

High Performance Benchmarks - Background

- **N Queens x 100,000**
  - Allocate 32B of memory
  - Access memory frequently to solve 8x8 N Queens board
  - Deallocate memory
  - High memory read/write access demonstrates interconnect bottleneck

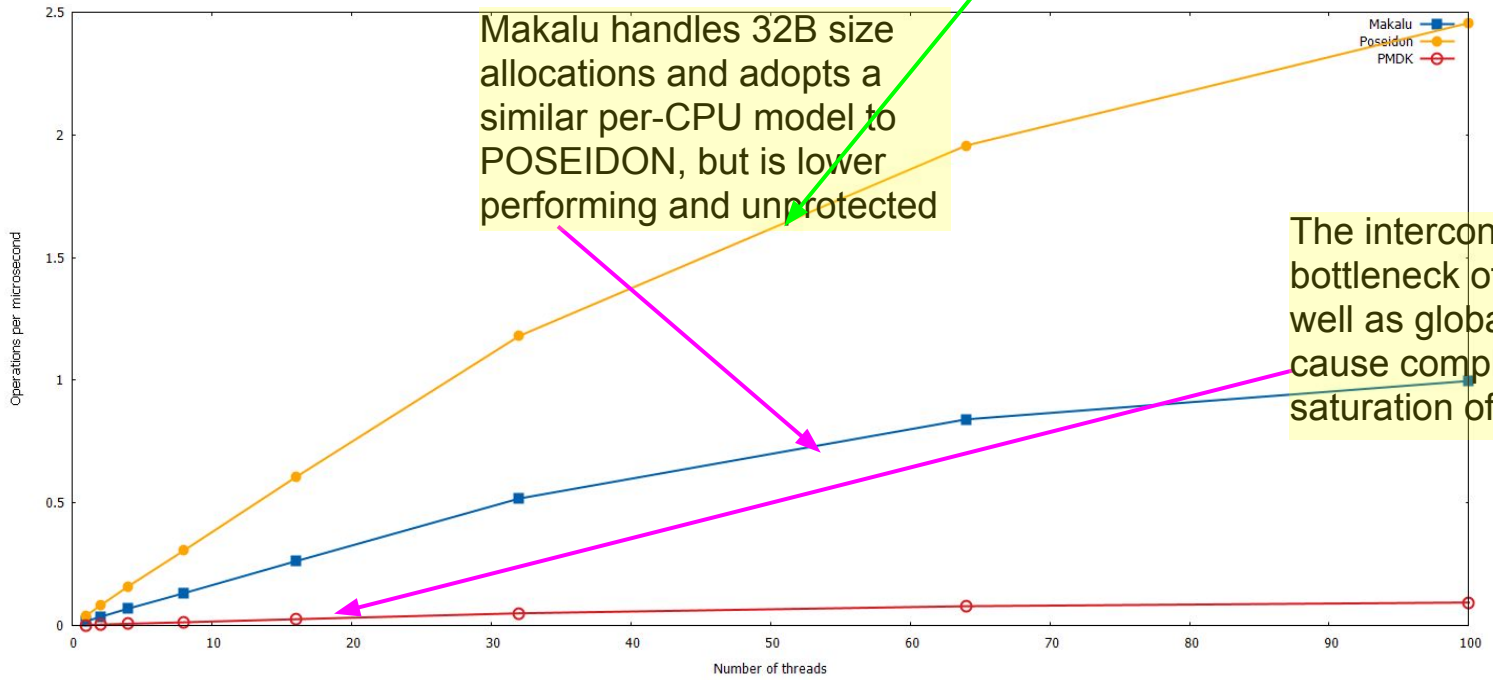
Evaluation

High Performance Benchmarks - N Queens

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

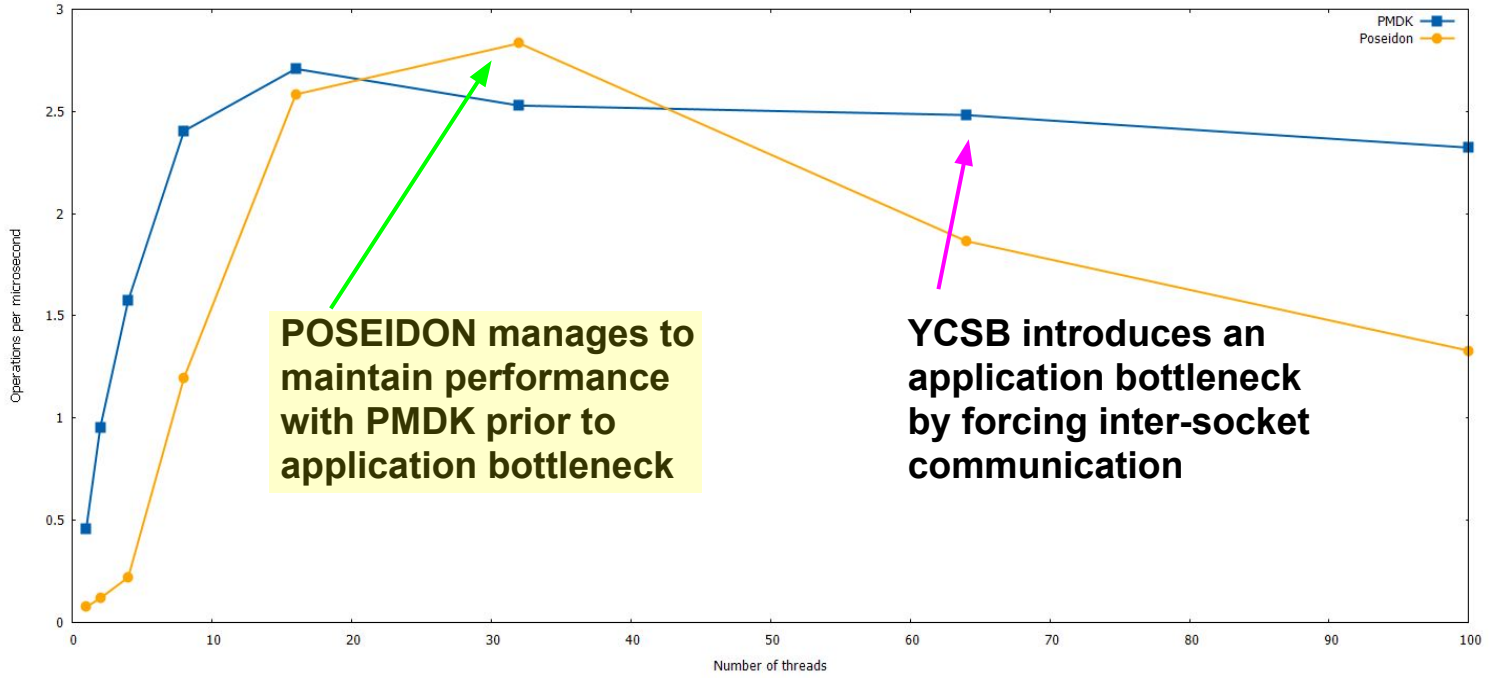
Makalu handles 32B size allocations and adopts a similar per-CPU model to POSEIDON, but is lower performing and unprotected

The interconnect bottleneck of PMDK, as well as global AVL tree, cause complete saturation of scalability



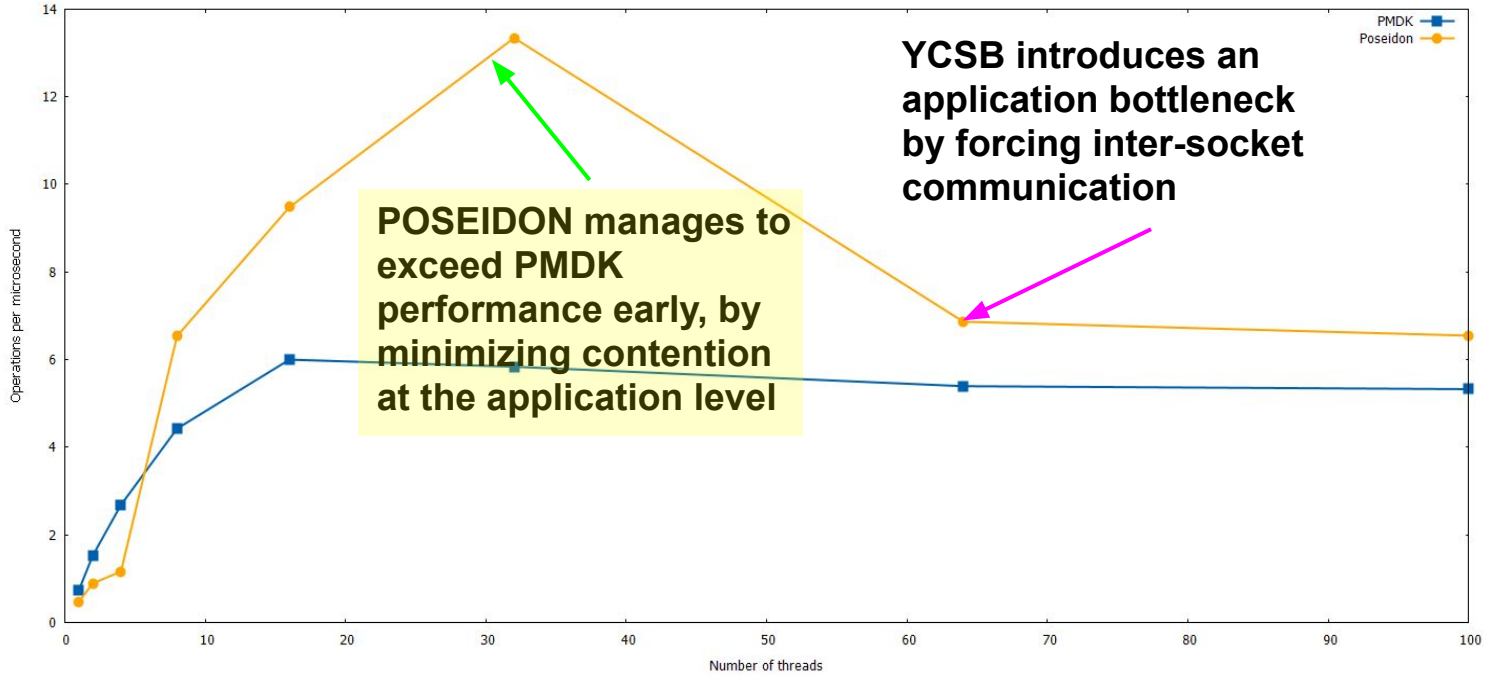
Evaluation

Real World Benchmarks - YCSB A (Update Heavy 50/50)



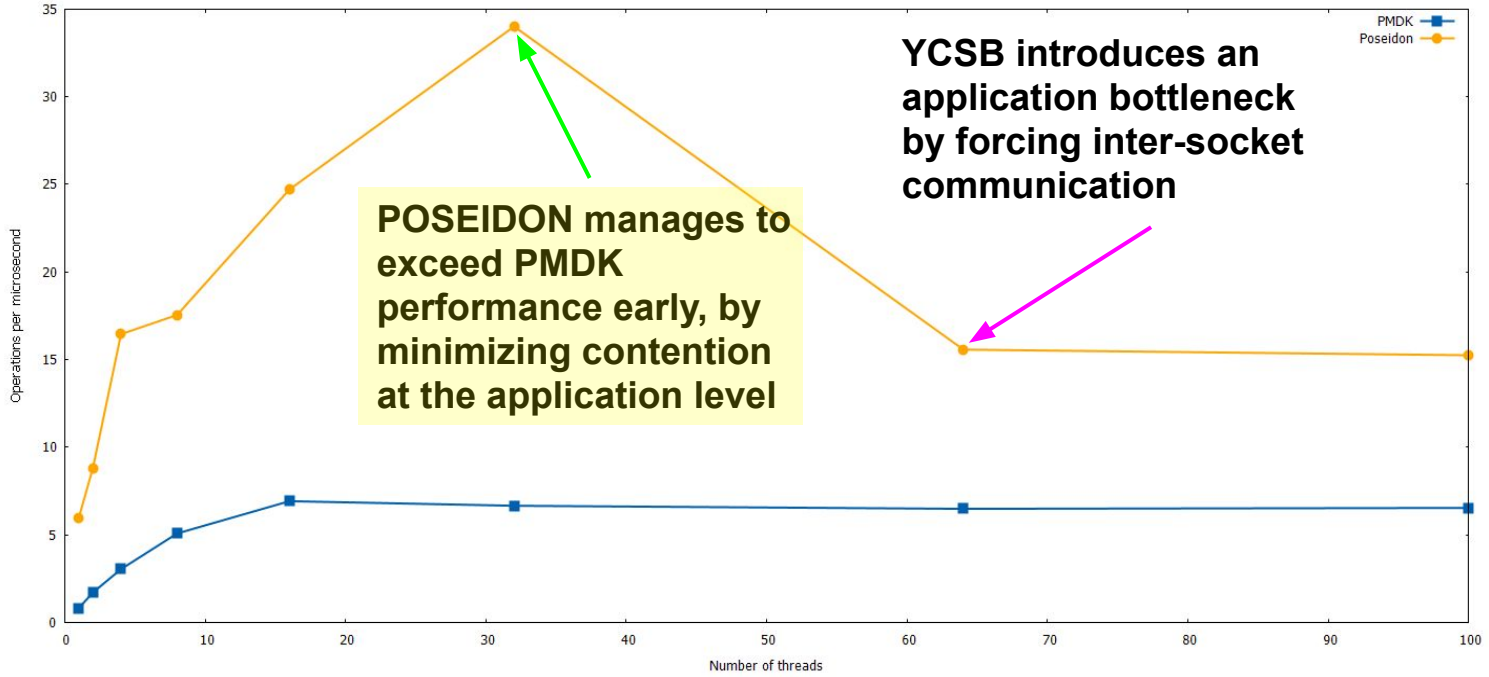
Evaluation

Real World Benchmarks - YCSB B (Read Heavy 95/5)



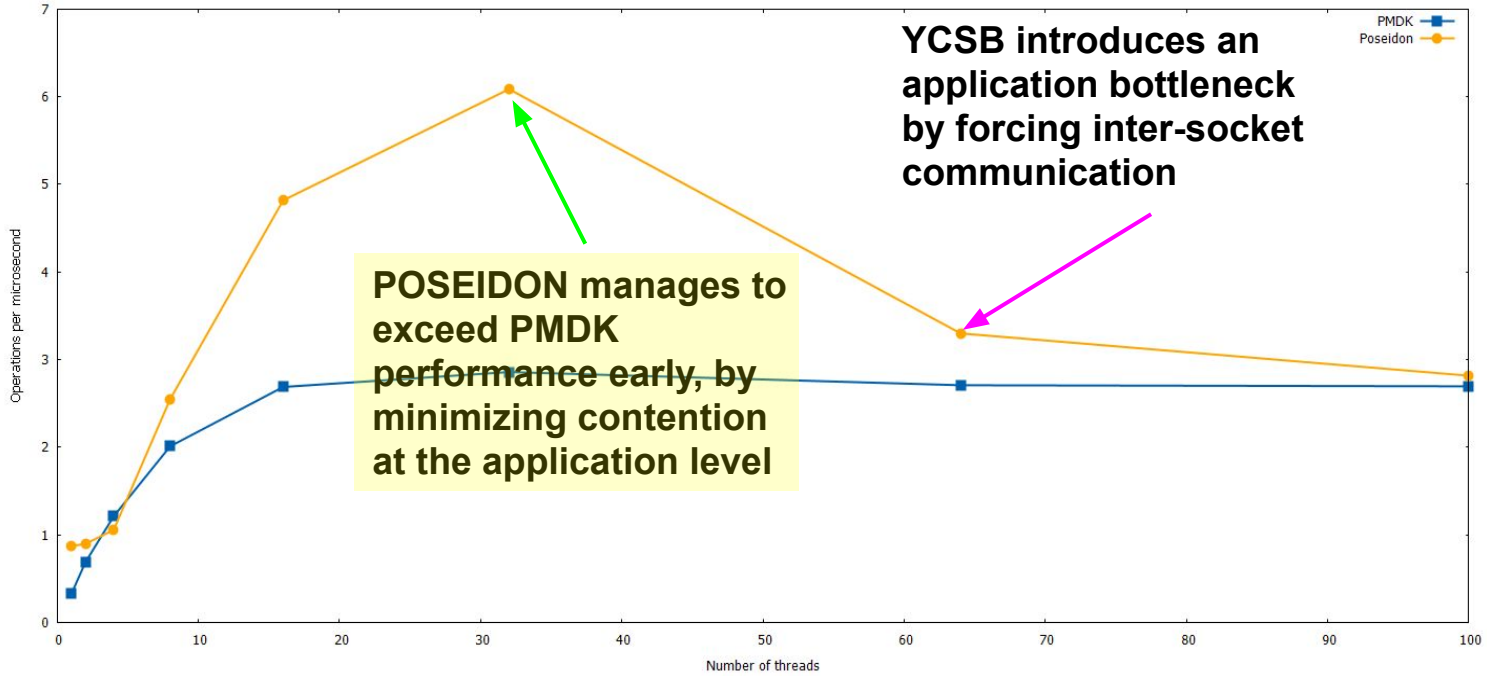
Evaluation

Real World Benchmarks - YCSB C (Read Only 100/0)



Evaluation

Real World Benchmarks - YCSB E (Threaded Conversations)



Evaluation

Real World Benchmarks - YCSB

- **Why** is YCSB introducing an application bottleneck?
  - The benchmark necessitates a **global heap**
  - Major restriction on maximizing memory controllers and minimizes inter-socket data transfer

- **How** can the bottleneck be mitigated?
  - More conscientious programming model
  - **POSEIDON**
    - Per-CPU sub-heap is critical