Poseidon: Safe, Fast and Scalable Persistent Memory Allocator

Anthony Demeri Wook-Hee Kim R. Madhava Krishnan
Jaeho Kim[†] Mohannad Ismail Changwoo Min
Virginia Tech [†]*Gyeongsang National University*

Abstract

Persistent memory allocator is an essential component of any Non-Volatile Main Memory (NVMM) application. A slow memory allocator can bottleneck the entire application stack, while an unsecure memory allocator can render applications inconsistent upon program bugs or system failure. Unlike DRAM-based memory allocators, it is indispensable for an NVMM allocator to guarantee its heap metadata safety from both internal and external errors. An effective NVMM memory allocator should be 1) safe, 2) scalable, and 3) high performing. Unfortunately, none of the existing persistent memory allocators achieve all three requisites. For example, we found that even Intel's de-facto NVMM allocator–`libpmemobj` is vulnerable to silent data corruption and persistent memory leaks resulting from a simple heap overflow.

In this paper, we propose Poseidon, a safe, fast, and scalable persistent memory allocator. The premise of Poseidon revolves around providing a user application with per-CPU sub-heaps for scalability and high performance, while managing the heap metadata in a segregated fashion and efficiently protecting the metadata using a scalable hardware-based protection scheme, Intel's Memory Protection Keys (MPK). We evaluate Poseidon with a wide array of microbenchmarks and real-world benchmarks, noting: Poseidon outperforms the state-of-art allocators by a significant margin, showing improved scalability and performance, while also guaranteeing heap metadata protection.

CCS Concepts: **• Software and its engineering → Allocation / deallocation strategies**.

ACM Reference Format:
Anthony Demeri Wook-Hee Kim R. Madhava Krishnan
Jaeho Kim[†] Mohannad Ismail Changwoo Min . 2020. Poseidon: Safe, Fast and Scalable Persistent Memory Allocator. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3423211.3425671

1 Introduction

Non-Volatile Main Memory (NVMM), such as Intel's Optane DC Persistent Memory [1, 24], is finally available to the public. Programs now can directly access NVMM without kernel intervention through `mmap`-ed NVMM memory by DAX-enabled file systems [5, 36, 37].

In this direct, NVMM-access programming model, NVMM space management is divided into two parts: First, DAX-enabled file systems manage the NVMM space of an NVMM device in a coarse-grained manner with files, and then persistent memory allocators manage the NVMM space of the said NVMM-backed file in a fine-grained manner. Persistent memory allocators [4, 14, 26, 30] provide memory allocation and free from/to an NVMM heap, similar to volatile memory allocators (*e.g.*, jemalloc [9] and TCMalloc [10]). In addition, most persistent memory allocators also provide transactional allocation and free of multiple NVMM objects, to support persistent transactions [14].

The design of persistent memory allocators is fundamentally different from volatile memory allocators in terms of safety guarantees. Persistent memory allocators must protect their heap metadata from any unfortunate accident, such as: program/system crash, sudden power outage, and program bugs. Persistent heap metadata contains important information on allocation status, such as allocation bitmaps and free memory chunk sizes, so, heap metadata corruption can cause silent user data corruption or persistent memory leaks. Furthermore, unlike DRAM, these problems can last forever, posing a serious threat to application security and stability. Existing persistent memory allocators [4, 26, 30], including Intel's `libpmemobj`, rely on logging to avoid heap metadata corruption from a crash but do not protect heap metadata from program bugs, such as heap overflow. Like volatile allocators, persistent memory allocators should provide high performance and multi-core scalability so they do not become a performance or scalability bottleneck for applications. In addition, persistent memory allocators should also provide effective memory capacity scaling, considering the current generation of NVMM hardware can scale up to 6TB (for a single two-socket system).

Unfortunately, we found that state-of-the-art persistent allocators, such as PMDK [15] and Makalu [4], neither guarantee heap metadata safety nor provide scalability. Additionally, we found that a simple programming bug, such as heap overflow, could *permanently* corrupt heap metadata of the PMDK allocator. With this metadata corruption, the PMDK allocator can allocate already-allocated memory, causing silent user data corruption, or it can fail to find a free memory chunk, causing a persistent memory leak. We will further discuss why existing persistent memory allocator designs fail to achieve heap metadata safety and scalability in the following sections (§2, §3).

In this paper, we aim to design a persistent memory allocator, named Poseidon that fulfills three essential requirements: (1) complete protection of NVMM heap metadata, (2) high performance, and (3) high scalability.

First, Poseidon should protect NVMM heap metadata completely from system and application crashes, misuse or malicious use of the memory allocator APIs (*e.g.*, double free, invalid free), and program bugs (*e.g.*, heap overwrite or underwrite). Existing designs guarantee crash consistency but fail to prevent metadata corruption from program bugs or misuse/malicious use of memory allocator APIs as shown in §3. To guarantee meatadata safety, Poseidon completely segregates the metadata and the user-data as opposed to the inline-metadata design in the existing allocators. Poseidon protects its segregated metadata by leveraging a new hardware feature – Intel Memory Protection Keys (MPK) [17, 27]. MPK can be used to enforce page-based protections in the user-space without any modifications to the page tables [27].

Second, Poseidon should provide high performance. The performance of an allocation is critical, as a persistent allocator is the central entity for managing NVMM space `mmap`-ed to a application's virtual address space. Hence, costs of protecting heap metadata and guaranteeing crash consistency should be minimal.

Third, Poseidon should be highly scalable, both in aspects of concurrency and capacity. The main advantage of persistent memory is large capacity, so any persistent memory operation should be constant-time as the NVMM capacity increases. Moreover, as multi-core systems are prevalent, high scalability to manycores is a fundamental requirement for all persistent memory allocators. Poseidon uses per-CPU sub-heaps design to achieve high performance and good multi-core scalability. The per-CPU sub-heap design reduces contention among the multiple threads and guarantees CPU local allocation to reduce the latency.

We make the following contributions in this paper:

- We found that the state-of-the-art persistent memory allocator provided neither guaranteed heap metadata safety nor multi-core scalability. In particular, we demonstrate that the PMDK persistent memory allocator (`libpmemobj`), which is the de-facto standard, can be easily and permanently corrupted due to a simple heap overflow bug.
- We introduce a new persistent memory allocator, named Poseidon. *To best of our knowledge, Poseidon is the first persistent memory allocator to guarantee heap metadata protection and achieve high performance and scalability in tandem.* Poseidon guarantees metadata safety using segregated heap metadata layout and efficient hardware-based memory access protection using Intel Memory Protection Keys (MPK). Also, Poseidon achieves high performance and scalability using per-CPU sub-heap design.
- We implemented Poseidon and evaluated Poseidon for micro-benchmarks and real-world applications. Our experimental results show that applications using Poseidon significantly outperform when compared to other state-of-the-art persistent memory allocators.

The rest of the paper is organized as follows. §2 provides the background. §3 is a case study for PMDK, which is a motivation for Poseidon's approach. §4 describes the overview of our approach and §5 explains the detail design of Poseidon. §6 shows Poseidon's implementation. §7 shows our evaluation results with micro-benchmarks and real-world applications. §8 discusses the saftey, correctness, and limitation of Poseidon and provides suggestions for PMDK. §9 compares Poseidon with previous research and §10 concludes.

2 Background

In this section, we describe how applications use NVMM, via NVMM-aware file systems and persistent memory allocators. We then describe the characteristics of and requirements for effective persistent memory allocators, specifically.

2.1 NVMM and Persistent Memory Allocator

There are two main software models, which grant userspace applications access to NVMM, as illustrated in Figure 1. The first software model is an *NVMM-aware file system*, so-called a DAX (direct access) file system. A DAX file system is typically an extension of a legacy file system (such as ext4-DAX or xfs-DAX), which bypasses page cache and directly accesses NVMM when serving file system operations. Applications can thus use standard file system APIs (*e.g.*, `open`, `read`, and `write`), tangentially acquiring the benefits inherent of a fast NVMM device. However, applications must still pay the price of file system overheads, additionally failing to leverage the desirable byte-addressability of NVMM. An additional software model involves using a *persistent memory allocator*, provided by a persistent memory library, such as Intel's

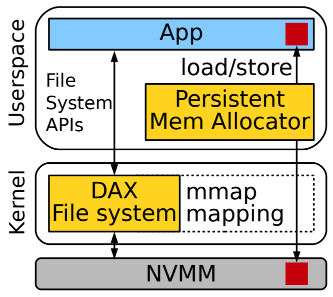

Figure 1. NVMM software layers

PMDK. A persistent memory allocator creates a persistent heap (which is essentially a large file on a DAX file system), used for storing persistent data. It manages the space of the persistent heap and provides `malloc/free`-like APIs. Thus, a userspace application can directly allocate persistent memory from the persistent memory allocator and access it using `load/store` instructions without suffering from the significant overhead of the storage stack. Hence, the role of a persistent memory allocator is, truly, critical in realizing the potential of NVMM-optimized software and underlying hardware.

2.2 Design Aspects of Persistent Allocator

We now discuss the general design characteristics of a persistent memory allocator, then contrasting such a design with the traditional volatile memory allocator design, so as to understand the challenges and requirements of a persistent memory allocator.

Persistent pointer representation. Persistent memory allocators manage one or more memory pools, which collectively encompass a persistent heap. When a persistent heap is loaded, memory pools are created (as necessary) and memory-mapped to an application's virtual address space. In order to represent a logical persistent-pointer, regardless of where a memory pool is mapped, across application or system restarts, persistent memory allocators [4, 14, 26, 30] typically adopt *16-byte persistent pointer representation*, which store a memory pool id and offset in the memory pool, such that an application converts a 16-byte persistent pointer to an 8-byte raw pointer prior to its direct use.

Transactional allocation. For a persistent memory allocator, the capability for performing transactional allocations is an additional requirement for supporting persistent transactions, which is a rather popular programming model in NVMM [7, 13, 20, 22, 34]. Within a persistent transaction, any NVMM write, including writes to a persistent memory allocator's internal metadata, must be atomically (all or nothing) updated to NVMM. Suppose that memory `P` and `Q` are allocated and then a crash happens before the transaction is persistently committed. The allocations of `P` and `Q` must be reverted, otherwise `P` and `Q` will be permanently leaked (as they will not be reachable from any other persistent pointers). When a system failure occurs during a transactional allocation, a persistent memory allocator must recover the metadata of the heap and deallocate appropriate space (which was previously allocated in the transactional allocation) by using an internal recovery protocol (*e.g.*, logging) [14, 26].

Crash consistency of heap metadata. When compared to a traditional, volatile memory allocator, we note that a persistent memory allocator requires additional metadata management efforts due to its non-volatility. Specifically, an allocator requires: 1) a root pointer and 2) crash consistency adherence [4, 14, 26, 30]. A *root pointer* resides at a well-known location in a persistent heap. Since an application cannot know the pointer value of a memory location in a memory pool across application or system restarts, a root pointer is used to provide an always-recoverable pointer.

For persistent memory allocators, it is additionally essential to guarantee the *crash consistency of heap metadata*, which contains critical metadata, such as allocation bitmaps and free memory space management information. If heap metadata is corrupted for some reason (*e.g.*, sudden power outage, program crash, etc), an application (or system) may not be able to load its persistent heap, thus losing all pertinent data. Note that traditional volatile memory allocators do not consider crash consistency, as DRAM is volatile. To guarantee crash consistency of heap metadata, most persistent memory allocators use a logging approach, which essentially makes a consistent copy of metadata before making any changes, logically similar to file-system journaling.

Protection of heap metadata from program errors. Besides an application crash, program bugs also can corrupt heap metadata. Suppose, for example, a program has a heap overflow bug or an integer overflow bug in calculating a persistent pointer. The program may then erroneously overwrite heap metadata with user data, thus, the heap metadata will have been corrupted; here, this is especially critical, as, once heap metadata is corrupted, its corruption is permanent, unlike a volatile memory allocator. In other words, while similar heap metadata corruption can happen for a volatile memory allocator, it does not cause any permanent data corruption; subsequently, protection of this metadata is expressly critical for any persistent memory allocator. In general, persistent heap metadata is vulnerable to program bugs since the metadata is simply part of a *read-writable* `mmap`-ed region [4, 14, 26]. Since there is no isolation between user data and metadata of persistent memory allocator, memory bugs in a program can permanently corrupt the persistent heap metadata.

We found that commonly-used metadata designs are vulnerable. For example, PMDK persistent memory allocator `libpmemobj` [14] adopts an in-place metadata design, placing metadata right before the allocated memory; a simple heap overflow bug can easily corrupt such in-place metadata. Makalu [4] adopts mark-and-sweep garbage collection to discover and fix persistent memory leaks without relying on logging. However, Makalu also fails to deal with metadata corruption. In particular, Makalu is vulnerable because if pointers in an object are corrupted, it can not reclaim any additional objects which are reachable by the object, similar to loss of a pointer in a linked list. We doubt if reachability-based garbage collection is practically useful in any memory unsafe languages like C and C++.

Scalability. While providing safety, persistent memory allocators should also maintain high performance and manycore

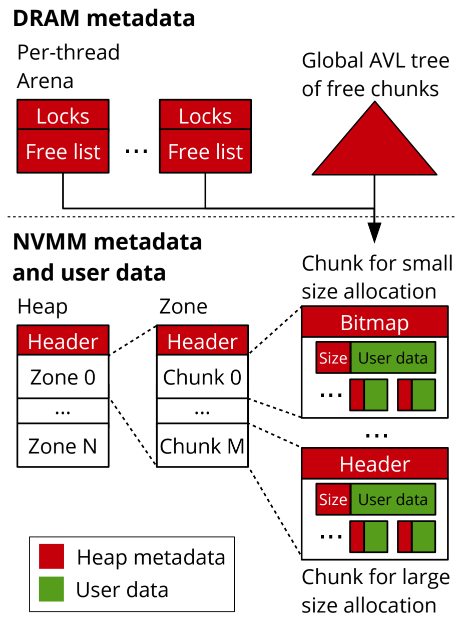

Figure 2. Architecture of PMDK persistent memory allocator. PMDK adopts in-place metadata design, which places metadata right before the allocated memory. Heap overwrite bugs can corrupt in-place metadata, resulting in permanent metadata corruption.

scalability. Since every memory access request is managed by a persistent memory allocator, it must not be a bottleneck on application (or system) performance. In addition, since commercially available NVMM provides a much larger capacity than that of DRAM, persistent memory allocators should also be scalable in memory capacity.

3 The Case of PMDK

In this section, we present our analysis on the de-facto PMDK allocator library, `libpmemobj` [16]. We show how a simple heap buffer overflow can corrupt the heap metadata permanently and eventually render the underlying application irrecoverable.

3.1 Design of `libpmemobj`

`libpmemobj` uses both NVMM and DRAM to maximize the performance. Since the read and write latency of DRAM is up to 5× faster than that of NVMM [18], `libpmemobj` stores frequently accessed data structures in DRAM, as shown in Figure 2. `libpmemobj` stores arena structures, free-lists, and an AVL-Tree in DRAM which are mainly used for finding free memory spaces. The AVL-Tree includes the information for allocations of large-size objects while the free-list includes the information for allocations of small size objects.

`libpmemobj` adopts in-place metadata layout; it stores metadata of the allocated object right before the allocated address. The in-place metadata essentially stores the size of allocation and allocation status. This design is useful to improve performance; since the metadata is located in a nearby space, `libpmemobj` can easily find metadata from a cache-line neighbor. Other information on memory chunk are stored in an allocation bitmap (small size) or a memory block header (large size). To guarantee the data consistency of the metadata, `libpmemobj` uses undo logging.

However, the design of `libpmemobj` is vulnerable to program bugs. There is no isolation between metadata and user data and also in-place metadata is stored near user data so a simple heap overwrite bug can easily corrupt in-place metadata, consequently corrupting the heap metadata [8, 28, 29, 39, 40]. We next show such permanent heap metadata corruption is possible in PMDK memory allocator due to heap overflow bugs.

3.2 Heap Metadata Corruption from Program Bugs

Our study of `libpmemobj` found out that there are three routes for heap metadata corruption (1) in-place metadata corruption in a user allocated memory, (2) direct metadata corruption, and (3) volatile cached metadata corruption in persistent memory heap.

In-place metadata corruption. In `libpmemobj`, the metadata of an allocated object is stored right before the allocated object as an object header, storing allocation size and status. The corruption of this in-place metadata can cause two irrecoverable problems. First, as shown in Figure 3, suppose a program has a heap memory corruption bug and it corrupts the size in the object header to larger value than its actual object size (Line 16). When `libpmemobj` frees the memory, it actually frees larger memory based on the corrupted size in header. Such metadata corruption eventually entails that `libpmemobj` allocates already-allocated memory (Line 28). If the program writes the data to the wrongly allocated memory, it silently causes permanent user data corruption. On the other hand, if the program corrupts the size in the object to smaller value as in Line 46, `libpmemobj` frees smaller memory based on the corrupted size in header. In this case, `libpmemobj` cannot reclaim the remaining space, causing a permanent persistent memory leak (Line 59). In-place metadata corruption is problematic in real-world applications because a heap overwrite bug can easily corrupt in-place metadata of adjacent objects, causing irrecoverable data corruption and/or permanent memory leak.

Direct metadata corruption. The metadata is stored in a static position of the NVMM space. For example, if we have a chunk for small-sized objects, the bitmap is stored at the beginning of the chunk. Since the chunk size is deterministic, it can be easily estimated where it stored. Hence, if a program bug modifies the bitmap directly, it can lose stored objects.

Volatile cached metadata corruption. Since `libpmemobj` uses both free-lists and an AVL-tree in DRAM, if the information in DRAM is corrupted so incorrect, memory allocation/deallocation can corrupt persistent memory heap. To

```
1   void pmdk_overlapping_allocation(nvm_heap *heap) {
2       void *p[1024], *free;
3       int i;
4
5       /* Make the NVMM heap full of 64-byte objects */
6       for (i = 0; true; i = ++i % 1024) {
7           if ( !(p[i] = nvm_malloc(64)) )
8               break;
9       }
10
11      /* Free an arbitrary object but before freeing
12       * the object, corrupt the size in its allocation
13       * header to larger number. It will make the PMDK
14       * allocator corrupt its allocation bitmap. */
15      free = p[i/2];
16      *(uint64_t *)(free - 16) = 1088; /* Corrupt header!!! */
17      nvm_free(free);
18
19      /* Since only one object is freed, the NVMM heap
20       * should be able to allocate only one 64-byte object.
21       * But due to the allocation bitmap corruption
22       * in the previous step, 9 objects will be allocated.
23       * Unfortunately, 8 out of 9 will be already allocated
24       * objects so it will cause user data corruption. */
25      for (i = 0; true; i = ++i % 1024) {
26          if ( !(p[i] = nvm_malloc(64)) )
27              break;
28          assert(p[i] == free); /* This will fail!!! */
29      }
30  }
```

```
31  void pmdk_permanent_leak(nvm_heap *heap) {
32      static void *p[INT_MAX];
33      int i, nalloc;
34
35      /* Make the NVMM heap full of 2MB objects */
36      for (i = nalloc = 0; true; i++, nalloc++) {
37          if ( !(p[i] = nvm_malloc(2*1024*1024)) )
38               break;
39      }
40
41      /* Free all allocated objects but before freeing objects,
42       * corrupt the size in their allocation header to smaller
43       * number. It will make the PMDK allocator corrupt chunk
44       * headers to smaller in size. */
45      for (i = 0; i < nalloc; i++) {
46              *(uint64_t *)(p[i] - 16) = 64; /* Corrupt header!!! */
47        nvm_free(p[i]);
48      }
49
50      /* Since all objects were freed, the NVMM heap should be
51       * able to allocate the same number of 2MB objects. But
52       * due to the chunk header corruption in the previous
53       * step, there is no such free chunk larger than 2MB.
54       * Thus, allocation will fail. */
55      for (i = 0; true; i++) {
56          if ( !(p[i] = nvm_malloc(2*1024*1024)) )
57              break;
58      }
59      assert(i == nalloc); /* This will fail!!! */
60  }
```

Figure 3. PMDK heap metadata corruption caused by heap overwrite. The corruption of in-place metadata in Lines 16, 46 can cause overlapping allocations (left) and permanent memory leaks (right). For brevity, we use the traditional `malloc`/`free`-like APIs instead of using PMDK's memory allocation APIs.

prevent this problem, we need to protect both DRAM layers and NVMM layers.

3.3 Non-scalable Performance

In spite of `libpmemobj` using a per-thread arena structure, it does not scale well; one primary reason for the non-scalable performance of `libpmemobj` is rebuilding a free-list on DRAM from NVMM. When `libpmemobj` deallocates small objects in NVMM, it unsets the bit in bitmap and does *not* put the deallocated space to the free-list in DRAM. When the free-list becomes empty, `libpmemobj` re-build the free-list by re-scanning the NVMM. Such rebuilding can happen frequently especially when persistent heap utilization is high. Moreover, free-list rebuilding process is sequential, preventing concurrent memory allocation/free operations.

Another reason for the poor scalability of `libpmemobj` is its global AVL-tree, used for indexing the larger free-blocks in the NVMM pool. When `libpmemobj` allocates a large memory space, it scans the AVL-tree in DRAM to find the appropriate free blocks. The global AVL-tree protected by a lock becomes a source of contention and hurts scalability of `libpmemobj` for large allocations.

4 Overview of Poseidon Architecture

We designed Poseidon aiming to provide (1) complete heap metadata protection from crash and program bugs, (2) high scalability as thread count increases, and (3) high performance without bottleneck in a critical path. In rest of this section, we discuss key design features of Poseidon (§4.1-§4.5), programming interface (§4.6), and how they fulfill aforementioned design goals (§4.7).

4.1 Per-CPU Sub-Heaps

As illustrated in Figure 4, a Poseidon heap consists of multiple per-CPU sub-heaps and a superblock maintains the list of sub-heaps. A sub-heap maintains its own logs for crash consistency, buddy list for allocation, information of each memory block, and its own lock for synchronization. Poseidon creates a sub-heap when the first memory is allocated on a CPU and places it on the NUMA domain of the CPU. Our per-CPU design not only reduce contention among multiple threads but also guarantees allocated memory is always NUMA local. Hence, a program does not need to pay remote NVMM costs across the NUMA domain in the common case. Note that cross-NUMA access overhead is more expensive in NVMM than DRAM [38]. Also, this allows multiple NVMM controllers (which are per-NUMA in x86 architecture) to be used, fully utilizing valuable hardware bandwidth.

4.2 Fully Segregated Metadata

Poseidon is the first persistent memory allocator that adopts fully segregated metadata design without performance and space overhead. Poseidon completely segregate heap metadata region and user-data region so there is *no in-place metadata* unlike other state-of-the-art persistent allocators (*e.g.*, PMDK). Superblock and per-CPU sub-heap metadata exists

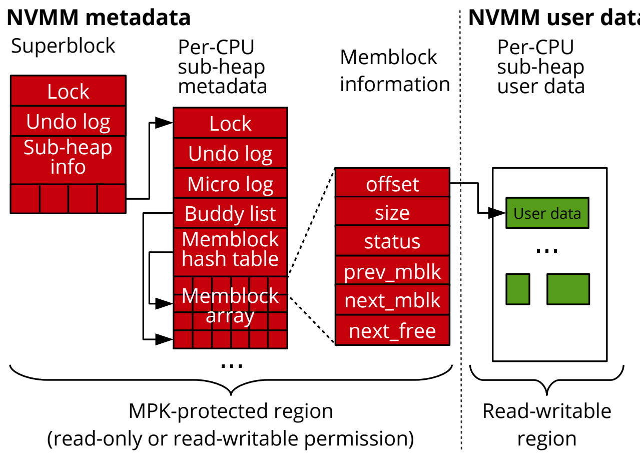

Figure 4. Heap layout of Poseidon persistent memory allocator. Poseidon completely segregate heap metadata region and user-data region, managing two regions with different access permissions. Poseidon grants write permission to the metadata region only performing allocation and free operations so a program cannot corrupt the metadata due to no access permission.

as a header, logically placed at the beginning of Poseidon persistent heap. The rest of area is used for user-data region, where a program actually accesses its allocated persistent objects. Due to the fully segregated metadata layout, Poseidon is able to prevent metadata corruption due to heap over-/under-writes, by maintaining separate memory access permissions for both metadata and user-data regions. We next explain how to efficiently manage access permission in Poseidon.

4.3 Metadata Protection with Intel MPK

By default, a Poseidon metadata region is not given a write permission. Only during an allocation/free operation, Poseidon temporarily grants a write permission (*i.e.*, read-writable) for the metadata region. Moreover, the write permission is given solely to the thread executing the operation. In other words, even if the write permission is granted for one thread, another thread cannot write to the metadata.

Poseidon uses Intel Memory Protection Keys (MPK) [17, 27] to efficiently change access permission of the metadata region for each thread. To the best of our knowledge, none of the other memory allocators, including secure memory allocators [3, 25, 31, 32], exploits Intel MPK to protect their metadata. When Poseidon initializes the heap, the entire metadata region is assigned to one of the 16 domains under a protection key, which is encoded in a page table entry. Changing the memory access permissions of the region is cheap; it only takes around 23 CPU cycles using `wrpkru` instruction [27]. Furthermore, such permission change is specific to a given thread, since the access permissions for a given key are stored in a register, which is, by definition, core-local; this prevents cross-thread metadata corruption.

Poseidon changes the permission of the metadata region to read-writable at the entry of allocation/free operations and reverts it back to read-only at the end of the operation. Notably, since MPK arrived prior to NVMM, all x86 processors that support NVMM also support MPK.

This design entirely prevents metadata corruption from program bugs, which has not been accomplished by any persistent memory allocator [4, 14, 26, 30]. Note that applying MPK protection to allocators using in-place metadata design is not feasible because MPK protection is per-page.

4.4 Metadata Management Using Hash Table

Poseidon manages information on both allocated and free memory blocks to perform defragmentation and protect metadata from double-frees and invalid-free bugs. Each memory block contains offsets, sizes, and statuses (*i.e.*, allocated or free). In addition, it contains the offsets of adjacent memory blocks (used for defragmentation) as well as the information for next free memory block in the buddy list (used for allocations). Accessing the memory block information is performance critical, as it is necessary for each malloc/free operation. Also, access performance must be scalable as the NVMM heap size grows. Poseidon uses a hash table to manage memory block information in constant time. To dynamically re-size the hash table, Poseidon uses a multi-level hash table, similar to one used in F2FS file system [19, 21]. The hash table in Poseidon also ensures the API cannot be maliciously used to corrupt metadata. Prior to any memory request (malloc/free), Poseidon uses the hash table to check if the memory address and its status are correct, so it prevents double-frees and invalid-free bugs, which can corrupt metadata.

4.5 Crash Consistency

Poseidon uses two logging schemes, namely undo logging and micro logging, to prevent metadata corruption from a program/system crash and sudden power outage. By using *undo logging*, Poseidon writes the original, unmodified metadata to the undo log prior to updating the metadata. This ensures partially written metadata can be restored to its original state, thus preventing metadata corruption upon a crash. In addition, Poseidon uses *micro logging*, which is a history of memory allocations, for transactional allocation (`poseidon_tx_alloc` in Figure 5). Upon transaction commit (*i.e.*, `is_end` of `poseidon_tx_alloc` is `true`), Poseidon truncates the micro-log. In other words, if the micro-log is not empty upon restart, the addresses in the micro-logs are allocated by an uncommitted transaction so Poseidon frees them to prevent persistent memory leak. The space overhead for logging is minimal because the undo log and micro log are truncated every successful allocation/free and successful transaction commit, respectively.

```
// Initialize a Poseidon heap with a given size and path name
heap_t *poseidon_init(const char *heap_path, size_t heap_size);
// Deinitialize a Poseidon heap
void poseidon_finish(heap_t *heap);

// Allocate an NVMM space with a requested size
nvmptr_t poseidon_alloc(heap_t *heap, size_t sz);
// Transactionally allocate a memory with a flag is_end
// denoting whether this is the last allocation in a transaction
nvmptr_t poseidon_tx_alloc(heap_t *heap, size_t sz, bool is_end);
// Deallocate an NVMM space pointed by ptr
void poseidon_free(heap_t *heap, nvmptr_t ptr);

// Convert an NVMM pointer to a raw pointer
void *poseidon_get_rawptr(nvmptr_t ptr);
// Convert a raw pointer to an NVMM pointer
nvmptr_t poseidon_get_nvmptr(void *p);

// Get the pointer of a root object
nvmptr_t poseidon_get_root(heap_t *heap);
// Set the pointer of a root object
void poseidon_set_root(heap_t *heap, nvmptr_t ptr);
```

Figure 5. Poseidon API

4.6 Programming Interface

Poseidon provides an intuitive APIs as shown in Figure 5. A program first need to initiate (or create) a Poseidon heap using `poseidon_init`. Upon `poseidon_init`, Poseidon loads (or creates) a superblock. Per-CPU sub-heaps will be loaded (or created) when the first malloc/free operation is performed on a CPU. The pointer to Poseidon heap can be freed using `poseidon_finish`. Besides a singleton allocation API (`poseidon_alloc`), Poseidon provides a transactional allocation API (`poseidon_tx_alloc`), in which allocated addresses are logged to a per-thread micro log. For a transactional allocation, a program should pass the flag (`is_end`), notifying whether the allocation is the last one or not. For the last transactional allocation in a transaction, a transaction is committed by truncating the micro log. Poseidon's persistent pointer type `nvmptr_t` holds 8-byte heap ID, 2-byte sub-heap ID, and 6-byte offset in a sub-heap. Poseidon provides pointer conversion APIs between a raw pointer and a Poseidon persistent pointer (`poseidon_get_rawptr`, `poseidon_get_nvmptr`). Similar to other persistent memory allocators, a program should convert a persistent pointer to a raw pointer before accessing the memory. Also, Poseidon provides APIs to get and set the root pointer of a heap (`poseidon_get_root`, `poseidon_set_root`) so a program can find NVMM objects from the root pointer.

4.7 How Poseidon Meets the Design Goals

Complete heap metadata protection. Poseidon provides complete heap metadata protection from (1) crash, (2) API misuse, and (3) program's memory corruption bugs. First, to protect metadata corruption from crash, Poseidon uses undo logging for a singleton allocation and micro-logging for transactional allocation. Next, Poseidon checks if a requested address and its status for free is correct by looking up the hash table of memory block information in a sub-heap. If Poseidon could not find the corresponding memory block (invalid-free bug) or its status is not allocated (double-free bug), Poseidon ignores the free request. Lastly, Poseidon completely segregates heap metadata from user data. The entire heap metadata is protected by an efficient hardware protection mechanism, Intel's MPK. The metadata region becomes read-writable only when running Poseidon code for a given thread. To best of our knowledge, Poseidon is the first persistent memory allocator that guarantees the complete heap metadata protection.

Manycore scalability. The Poseidon heap consists of per-CPU sub-heaps to minimize synchronization costs of concurrent malloc/free operations, while guaranteeing NUMA-local allocations, to fully utilize scarce hardware resources (especially memory controllers).

High performance. Poseidon maintains information of all allocated/free memory blocks to prevent double-free and invalid-free bugs. Thus, every Poseidon operation requires to look up and change the memory block information. Poseidon does this in constant time by using a multi-layer hash table to manage memory blocks, rather than using tree structures as in other persistent memory allocators [14, 26]; thus, regardless of the pool size or allocation size, allocation and free time is constant.

5 Design of Poseidon

5.1 Loading the NVM Heap

Upon initialization of Poseidon, an NVMM heap is loaded. In the NVMM heap loading phases, an MPK protection key for metadata is allocated. After that, Poseidon makes sure that its metadata is a consistent state by checking its own internal undo log. Next, Poseidon iteratively maps all existing sub-heaps to persistent memory and maintains their sub-heap pointers locally. Now, both the superblock and sub-heaps are mapped to persistent memory. To protect this memory region, Poseidon allocates an appropriate MPK key and protecting all metadata regions from external write access. After making the memory region safe, Poseidon makes each sub-heap consistent state by processing their respective undo logs and micro logs. Finally, all metadata locks are unlocked, and the NVM heap has been successfully loaded.

5.2 Singleton Allocation

When a user requests a standard allocation (`poseidon_alloc`), Poseidon begins by changing the permissions of heap metadata to read-writable using MPK. After that, Poseidon chooses a sub-heap where the requesting thread is running. Poseidon manages free memory blocks using a buddy list, which is an array of free lists where each list contains only a certain size class of free blocks. Hence, based on the requested allocation size, Poseidon accesses the sub-heap's internal buddy list and finds a large enough free block. If the free block is too large, Poseidon splits the free block into two new free blocks. When there is no available free block in the buddy list, Poseidon performs defragmentation, which we will discuss shortly. If a large enough free block is found, its memory block status is updated in the hash table. If a collision is detected during hash table indexing, first, linear probing is used. If this determines no available blocks exist within a probing range, defragmentation of the hash table will occur. Still, if defragmentation does not provide an available index, the hash table will be extended (via a multi-level hash table). Note that Poseidon performs undo logging for the metadata before modifying its original data. Poseidon updates the original metadata after the persistent barrier (cache line flush followed by memory barrier in x86 architecture) of the undo logging. When all changes in the original metadata is persisted, Poseidon atomically truncates the undo log. Finally, Poseidon changes the metadata permission to the read-only via MPK.

5.3 Transactional Allocation

In a transactional allocation (`poseidon_tx_alloc`), all internal metadata is updated identically to the standard allocation for each sub-allocation, with the addition of persisting the allocated address in an internal micro-log before truncating the undo log. After persisting the micro-log, Poseidon truncates the undo log for metadata. Lastly, Poseidon atomically truncates the internal micro-log as a commit point of a given transaction (`is_end == true`).

5.4 Defragmentation

Poseidon triggers defragmentation of a sub-heap when (1) there is no free memory block in the requested size class, or (2) there is no space in the hash table after performing linear probing. In the first case, Poseidon iterates free blocks in buddy lists whose size is smaller than the requested size and tries to merge left and right adjacent blocks to form a larger free block. In the second case, Poseidon iterates all free memory blocks within the linear probing space and tries to merge left adjacent blocks to form a larger free block and a free space in the linear probing space. By using this approach, defragmentation is performed at a local level, which is a both scalable and high performance, contrary to the design of existing persistent memory allocators, which perform defragmentation on a global size-independent level [26] or a global level [4].

5.5 Deallocation

Memory deallocations from a sub-heap follow a near-inverse process to that of memory allocations. First, Poseidon acquires the write permission to the metadata using MPK. Then, based on the given memory address to free, Poseidon searches the hash table on an appropriate sub-heap, using the address as a key. If the address is found, Poseidon performs undo logging for the metadata of the memory block. After persisting the undo log, Poseidon changes the memory block status to free and insert it to the tail end of its respective size class of the buddy list to prevent immediate reuse of the allocation. When all metadata updates successfully persisted, Poseidon truncates undo log as a commit point of the deallocation. Finally, Poseidon changes the sub-heap metadata permission to read-only using MPK. Note that Poseidon supports preventing invalid free and double free. If the address is not found in the hash table, the free is an invalid free. If the status of the address is in a free status, the free is a double free. Thus, these frees are both rejected.

5.6 Space Management of Heap Metadata

Poseidon reduces its metadata footprint by *hole-punching* (using the `fallocate` syscall) unused metadata pages and returning them to the underlying filesystem. Poseidon primarily uses this technique to eliminate unused levels of its multi-level hash table. When Poseidon needs more memory for metadata, it first accesses a "hole-punched" region of memory. In this case, Poseidon needs only write to its known address because the region has already been mapped then underlying filesystem will allocate an NVMM page. If there is no space in the hole-punched region, Poseidon extends the mapping of the metadata region. With this, Poseidon utilizes the hash table memory only when needed.

5.7 Synchronization

Poseidon protects a sub-heap from concurrent access by acquiring the lock of a per-CPU sub-heap. Therefore, each sub-heap can be concurrently accessed without any interference. Due to the per-CPU design, usually there is no contention in accessing a sub-heap. The contention happens only when a thread running on CPU `X` tries to free a memory in a sub-heap belonging to CPU `Y`. In this case, a local thread may contend with non-local treads performing free. However, we empirically found that such contention happens very rarely so it does not harm scalability.

5.8 Crash Consistency and Recovery

Upon initialization of Poseidon, all undo logs and micro logs are checked for consistency of metadata. In Poseidon, a successful allocation/deallocation results in the truncation of both undo logs and micro logs. It means that the presence of data within either log indicates a crash occurred. When an undo log is not empty, Poseidon performs crash recovery by reverting partial changes within the log to restore the heap metadata. To do this, Poseidon first changes the permissions of the heap-metadata to have read-write access using MPK. After that, the contents of the undo log are copied to the original metadata location and persisted to storage. When the original metadata is recovered and persisted, Poseidon truncates the undo log atomically. If a micro log is not empty, Poseidon deallocates all addresses in the micro log then

Poseidon truncates the micro log. Since Poseidon's recovery process truncates both undo logs and redo logs when it is successfully completed, Poseidon can still detect whether a crash happens. If a crash occurs during the recovery process, Poseidon will replay both the undo log and micro log again and truncates them. Note that replaying logs are idempotent, so it does not affect any consistency of data even if Poseidon replays the logs multiple times. Finally, Poseidon sets permission of the metadata to read-only and completes the recovery process.

6 Implementation

We implement Poseidon in C, comprising around 16,000 lines of code and 18,000 lines of associated unit tests. Since there are few open-source persistent memory allocators, and it is non-trivial to separate in-place metadata designs from existing, open-source persistent memory allocators, we wrote Poseidon from scratch. To guarantee the persistence of Poseidon's data in NVMM, we use the `clwb` instruction with `sfence`, which efficiently flushes CPU cache lines to NVMM.

7 Evaluation

We evaluate the scalability of Poseidon compared to PMDK [14] and Makalu [4] using a microbenchmark and a simulated server workload (§7.2, §7.3). We then show the performance impact of real-world applications with Poseidon, to demonstrate the impact of Poseidon (§7.4, §7.5).

7.1 Evaluation environment

We evaluate the performance of Poseidon using a system that consists of 2 socket, 56 cores (112 logical cores) Intel Xeon Platinum 8280M CPU, 768GB DRAM, and 3.0TB (12 x 256GB) of Intel Optane DC Persistent Memory (DCPMM). Note that since the current generation of Intel Optane DCPMM supports only up to two socket machines, two sockets are the maximum number of the socket that we can conduct the experimental evaluation as for now.

Our experiments were conducted on the Linux 4.19.0 kernel. Regarding the software being evaluated, we use the most recent stable versions for both persistent memory allocators as of this writing (PMDK v1.7). We do not include PAllocator [26] because its source code is not publicly available.

7.2 Microbenchmark

We first ran a microbenchmark that performs allocations and frees in random order to quantitatively show the performance gain from the per-CPU sub-heap, multi-layer hash table based metadata management, and low latency MPK based metadata protection in Poseidon. The microbenchmark performs 100 allocations, and 100 frees in a random order, repeating this process such that a total of 1 million allocations/frees, with varying numbers of threads and allocation sizes, are performed. The microbenchmark does not perform any inter-thread free to show the ideal maximum performance.

Poseidon. As Figure 6 shows, Poseidon significantly outperforms all persistent memory allocators by upto 60× and shows linear scalability upto 64 threads. Poseidon is able to maintain scalability by using per-CPU sub-heap contained metadata. Moreover, constant time for memory allocation/free comes from hash-table based metadata management, which tangentially improves the scalability of Poseidon. Finally, our MPK based metadata protection is able to provide safety at low latency. So Poseidon shows better performance and scalability against other persistent memory allocators nearly for all contention levels while providing metadata protection.

Makalu. Makalu [4] shows poor scalability, as both the allocation size increases and the number of threads increases. The main reason for its poor scalability is its metadata design, which incorporates a *global chunk list* for allocations greater than 400 bytes. When an allocation size is larger than 400 bytes is requested, a global lock becomes a scalability bottleneck. Furthermore, it also places additional global locking constraints on allocations less than 400 bytes by using a *global reclaim list*. The global reclaim list, which maintains a list of free blocks to be distributed among thread-local free-lists, grows when a given thread's local free-list has a large number of free-list blocks available. In other words, when a thread's local free-list has a large number of free blocks available, it adds them to the global reclaim list (which requires global locking). So, when performing 100 allocations and 100 frees, even with a block size of 256 bytes, we observe a scalability bottleneck due to the global reclaim list; for example, in Figure 6, with block size less than 400 bytes, we observe a 6× performance loss, but, with a block size greater than 400 bytes, we observe greater than a 1000× performance loss.

PMDK `libpmemobj`. The de-facto standard persistent memory allocator PMDK [14] shows better scalability than Makalu. However, it also saturates the performance when the number of threads is larger than 32 regardless of the allocation sizes. The main reason for the saturation of performance is the internal design of PMDK, as shown in §3. In PMDK, a given heap contains 12 *arenas*, each of which has its own lock and accesses a global AVL tree of free memory chunks, rather limiting each arena to its own metadata. As such, access to a given region within the *global AVL tree* necessarily introduces a scalability bottleneck, particularly observed when the number of threads begins to outgrow the number of arenas. In addition, PMDK introduces an additional implementation bottleneck to their system by using a *global action log* to batch free operations together. This design helps amortize the overhead involved in flushing data to persistent memory. However, for free-heavy applications, this action log becomes a source of contention as clearly observed in

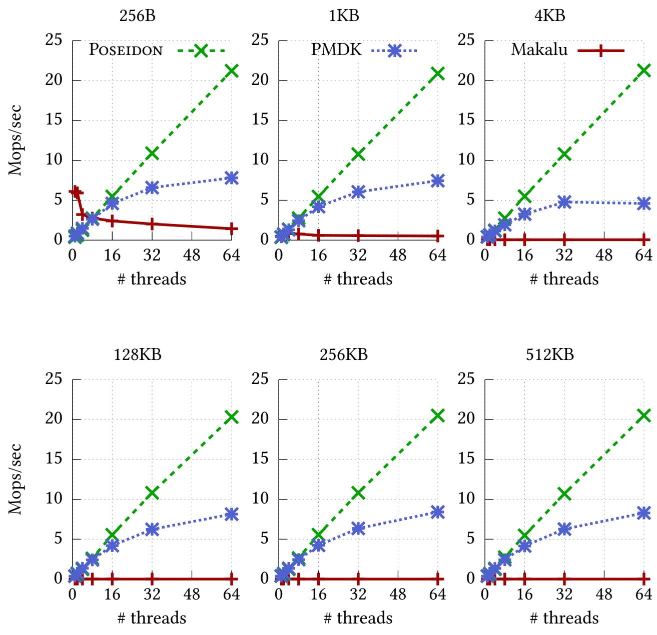

Figure 6. Performance of a pair of 100 mallocs and 100 frees in random order with different allocation sizes.

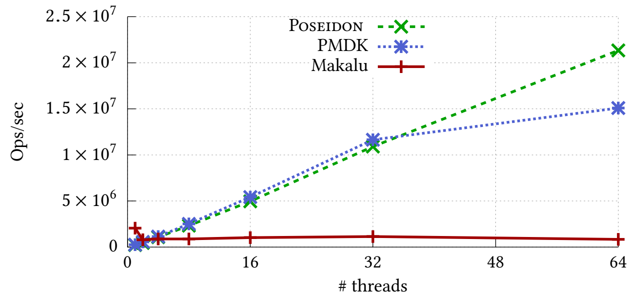

Figure 7. Performance of Larson benchmark.

Figure 6, where PMDK exhibits inverse performance as the number of threads surpasses 16.

7.3 Larson benchmark: Real-World Server Allocation Pattern

The Larson benchmark [2] simulates a server: performing multiple, concurrent, cross-thread allocations, and deallocations. Larson benchmark generates `N` objects of allocation/deallocation while varying the size of objects randomly. We run the Larson benchmark for 10 seconds with a varying number of threads, as in the PAllocator [26]. Similar to the results of our microbenchmarks, Poseidon significantly outperforms other persistent memory allocators up to 4×. Once again, this is due to the bottlenecks which are observed and noted in §7.2.

7.4 Real World Application: High Performance Benchmark

To further demonstrate Poseidon's performance and scalability, we provide real-world examples of allocator use in computation-intensive applications. In these high-performance benchmarks, we evaluate Poseidon, Makalu, and PMDK in real-world scenarios, which involve heavy manipulation of allocated memory regions. We use the following benchmarks: the *Ackermann* benchmark, the *Kruskal* benchmark, and the *N Queens* benchmark. We run all of these benchmarks while varying the number of threads.

Ackermann benchmark. We perform a single, 1GB allocation, which is filled in with Ackermann results upto (4, 5). The 1GB region is allocated, utilized as a cache to compute Ackermann results, deallocated, and repeated 100,000 times. Poseidon performs upto 242× better than Makalu and 6.4× better than PMDK, respectively. We similarly observe the bottlenecks previously discussed due to the design constraints of both Makalu and PMDK.

Kruskal benchmark. We solve Kruskal Minimum Spanning Tree (MST) implementations of order 5, each of which performs three allocations of 512 bytes before solving the MST, deallocating the memory, and repeating the process 100,000 times. Makalu shows better performance when the number of threads is less than eight because Makalu's crash consistency protocol does not rely on the logging scheme. Instead, it uses the mark-and-sweep garbage collection. Moreover, the main scalability limitations come from global metadata management structures. So, Makalu shows better performance when the number of threads and allocations are small. However, when the number of threads is larger, the performance gaps between Poseidon, Makalu, and PMDK are much larger. Overall, Poseidon outperforms PMDK, Makalu, by up to 4.3×, 16.6×, respectively.

N Queens benchmark. We solve N Queens puzzles of board size 8 using one 32-byte allocation, which is deallocated when the N Queens puzzles are complete. Similar to the Ackermann and Kruskal benchmarks, this process is repeated 100,000 times. The performance of Poseidon is 2.3× and 24.8× better than Makalu and PMDK, respectively. Interestingly, Makalu shows better performance than PMDK, because of delayed mapping of memory which maps the memory closer to the running thread's socket. PMDK creates pools in the main thread, which maps memory potentially farther from a running thread's socket. Thus there is a high possibility of failing to utilize the maximum number of memory controllers and increasing socket interconnect contention.

7.5 YCSB benchmark

The YCSB benchmark [6, 35] simulates a key-value store scenario, performing multiple, concurrent, cross-thread allocations, and deallocations in a persistent index structure. We

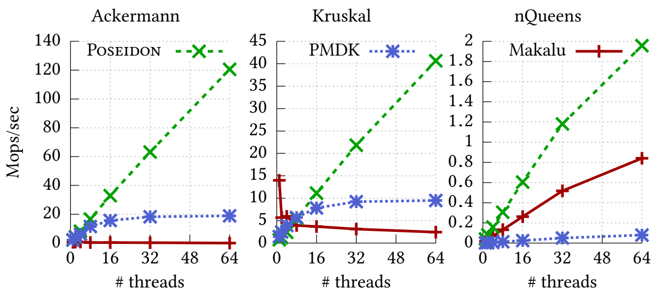

Figure 8. Performance of real-world, high performance benchmark.

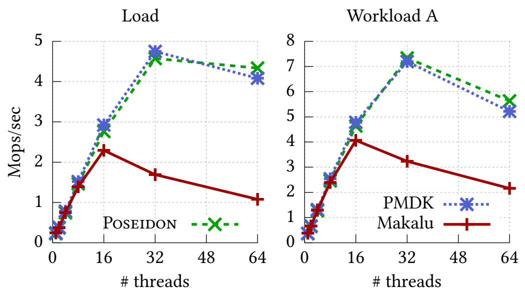

Figure 9. Performance of YCSB benchmark.

modified FAST-FAIR B+-tree [12], a persistent byte-addressable B+-tree to use Poseidon, PMDK, and Makalu persistent memory allocator. In this evaluation, we first load 10 millions of keys into the B+-tree and execute 10 millions of operations for `Workload A`. Note that, since YCSB workloads are mostly read-intensive workloads, we choose memory allocation heavy workload `Load` and `Workload A`. As Figure 9 shows, Poseidon's performance mirrors that of PMDK. The interesting point is that even though Poseidon uses fully segregated metadata, it shows similar or slightly better performance than PMDK. Moreover, Poseidon guarantees full protection of heap metadata from program bugs. Note that the performance of the YCSB benchmark mainly depends on the performance of the index structure. For example, when we execute `Workload A`, the YCSB benchmark first needs to traverse the B+-tree internal nodes to find the proper leaf node (data node). This traverse operation introduces the overhead to the YCSB benchmark, so the memory allocation overhead is relatively smaller than previous evaluations. Moreover, since the bandwidth of Intel Optane DCPMM is limited [38], both Poseidon and PMDK shows performance degradation when the number of thread is larger than 32. Makalu shows similar performance to Poseidon and PMDK upto 16 threads. However, when the thread count is larger than 16, the performance significantly degraded due to its non-scalable metadata design.

8 Discussion and Limitations

Safety and correctness. Poseidon enforces the safety invariant such that the heap metadata is written only inside the Poseidon code, where we grant write permissions during its execution. Even if a writer accidentally spawns (forking a new process) or triggers other threads unintentionally, the other thread/process cannot compromise the heap metadata because MPK is per-CPU (i.e., per-thread). In terms of correctness, Poseidon is built on top of undo and micro logging which are widely used for transaction recovery and already well-proved regarding correctness.

Mitigating metadata corruptions by program bugs in PMDK. Since PMDK follows the in-place metadata design, it is hard to adopt Intel MPK for PMDK because MPK projection has per-page (4KB) granularity. While we think PMDK's in-place metadata design, storing heap metadata in NVMM user data area, is fundamentally susceptible to metadata corruption by program bugs, there is an urgent need to harden de-facto standard PMDK allocator. One mitigation approach is adding a canary value to its in-place metadata structures; when freeing an NVMM memory, a persistent allocator checks if its in-place metadata is corrupted by checking the canary value. If the canary is corrupted, the allocator can skip freeing a memory so as not to further propagate metadata corruption. While this neither guarantees the metadata protection nor prevents persistent memory leak, it can mitigate the side effect of in-place metadata corruption by stopping the propagation of its corruption.

Limitations. One limitation is that Poseidon cannot guarantee NVMM heap metadata protection from the malicious use of Intel MPK. In particular, if an attacker can hijack program control flow and execute a non-privileged instruction `wrpkru` to change the permission of the heap metadata region to read-writable permission, she will succeed in manipulating Poseidon metadata. While this kind of security attack is out of scope of this paper, we can prevent such attacks by adopting binary inspection to get rid of potential misuse of MPK, similar to Hodor [11] and ERIM [33].

Another limiation is that current our Poseidon uses multi-level hash table for the memory management. The multi-level hash table is good enough to support the Intel Optane DCPMM capacity (3 TB) on our evaluation systems. However, when the capacity of NVMM is much larger than that we used in this paper, our Poseidon prototype can be further optimized for the huge capacity by using a more advanced index scheme.

9 Related Work

Persistent memory allocators. Recently, many studies for persistent allocators have been conducted to achieve scalability and persistence on many-core systems. However, we observe that none of allocators are scalable on a large number of cores and guarantee complete metadata protection. PMDK allocator `libpmemobj` [14] maintains per-thread arena structure for high scalability but, as we discussed, the limited number of per-thread arena, a global AVL tree for free chunks, and reloading free list become a performance and scalability bottleneck. More importantly, PMDK allocator is vulnerable to permanent metadata corruption particularly due to its in-place metadata design.

Makalu [4] suffers from limited manycore scalability due to several global metadata management structures. Makalu's crash consistency protocol relies on the mark-and-sweep garbage collection to discover and fix the persistent memory leak. However, the mark-and-sweep garbage collection approach is vulnerable to metadata corruption because if pointers in an object are corrupted, it will not be able to reclaim additional objects which are reachable by the objects. Truly, it is questionable if reachability-based garbage collection is practically useful in memory unsafe languages like C and C++.

PAllocator [26] attempts to provide scalability by maintaining small and big object allocators per core but it still has a problem with a large number of cores because it uses tree structure to manage its memory allocation information as in PMDK. Also, PAllocator does not guarantee metadata safety from program bugs.

nvm_malloc [30] is a persistent allocator based on jemalloc [9]. nvm_malloc supports safe NVMM allocations with two distinct steps: reserve and activation. Applications create persistent links to the allocated region before updating metadata to track objects. It also does not provide metadata safety from program bugs.

Scalable memory allocators. The majority of persistent memory allocators are influenced by scalable memory allocators designed for DRAM. Hoard [2] improves performance of applications by maintaining per-processor heaps so as to reduce contention. TCMalloc [10] provides fast memory allocation by using thread-local cache when allocating small size objects, otherwise using fine grained locking to manage a central heap. jemalloc [9] is a scalable memory allocator which is widely used in cross-platform applications. It was specifically designed for minimal memory fragmentation and multi-threaded concurrency support. SSMalloc [23] is another scalable memory allocator focused on achieving low latency with lock-free synchronization.

Secure memory allocators. State-of-the-art secure allocators [3, 25, 31, 32] mostly adopt a segregated metadata layout and rely on some form of randomization to bolster their security. Unfortunately, none of them are designed for NVMM, and suffer from high runtime and memory overhead. None of these use MPK, like Poseidon does, to protect heap metadata, so none of them can truly guarantee metadata protection. DieHard [3] proposes the notion of probabilistic memory safety, which is an ideal but unimplementable runtime system that provides infinite heap semantics. DieHard uses probabilistic guarantees to avoid memory errors. DieHarder [25] randomly allocates pages over the entire possible address space, and carves them up into size-segregated chunks tracked by an allocation bitmap. Both techniques suffers from a high overhead of up to 40% on allocation intensive benchmarks. FreeGuard [31] attempts to combine techniques from both BIBOP (Big Bag of Pages) and freelist allocators. The main approach is acquiring a large block, which is then divided into multiple sub-heaps. FreeGuard uses a per-thread sub-heap design. Guarder [32] is similar in design to FreeGuard, but is mainly concentrated on randomization-entropy and tunable security guarantees [32]. Guarder introduces allocation and deallocation buffers, which choose objects randomly on allocation. Unfortunately, both Guarder and FreeGuard suffer from high memory overheads of up to 37%.

10 Conclusion

We presented Poseidon, a safe and scalable persistent memory allocator. To the best of our knowledge, Poseidon is the first persistent allocator guaranteeing complete metadata protection. We illustrated how the existing defacto memory allocator (PMDK) is vulnerable to silent data corruption and persistent memory leaks. In order to ensure metadata protection, Poseidon demonstrated its management of heap metadata in a segregated fashion, guarded from both internal and external errors by using MPK. To achieve better scalability and performance, Poseidon was shown to utilize per-CPU sub-heaps, managing sub-heap metadata by hosting the buddy list and free blocks of respective hash tables on the same CPU. Critically, we evaluated Poseidon against the state-of-art memory allocators, such as PMDK and Makalu, with microbenchmarks, real-world, computation-intensive applications, and NVMM-optimized persistent index with YCSB benchmark. For all cases, Poseidon shows seamless scalability and superior performance, as opposed to its counterparts, while simultaneously being the only persistent memory allocator also guaranteeing vital metadata safety.

Acknowledgement

We thank our shepherd Yu Hua and the anonymous reviewers for their insightful comments. This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2014-3-00035)

References

[1] Anandtech. 2018. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here

[2] Emery D Berger, Kathryn S McKinley, Robert D Blumofe, and Paul R Wilson. 2000. Hoard: A Scalable Memory Allocator for Multithreaded

Applications. In *Proceedings of the 9th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Cambridge, MA, 117–128.

[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Ottawa, Canada, 158–168.

[4] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. 2016. Makalu: Fast Recoverable Allocation of Non-volatile Memory. In *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, Amsterdam, Netharlands, 677–694.

[5] Dave Chinner. 2015. xfs: updates for 4.2-rc1. https://lwn.net/Articles/635514/

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)* (Indianapolis, Indiana, USA). ACM, Indianapolis, Indiana, USA, 143–154.

[7] Andreia Correia, Pascal Felber, and Pedro Ramalhete. 2018. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the ACM symposium on Parallelism in algorithms and architectures (SPAA)*. Vienna, Austria, 271–282.

[8] Moritz Eckert, Antonio Bianchi, Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2018. Heaphopper: bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, 99–116.

[9] Jason Evans. 2006. A Scalable Concurrent malloc (3) Implementation for FreeBSD. In *Proceedings of the BSDCan*. Ottawa, Canada.

[10] Google. 2007. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[11] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA, 489–503.

[12] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-addressable Persistent B+-tree. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*. Oakland, California, USA, 187–200.

[13] Intel. 2016. C++ bindings for libpmemobj (part 6) - transactions. http://pmem.io/2016/05/25/cpp-07.html

[14] INTEL. 2019. Persistent Memory Development Kit. http://pmem.io/

[15] INTEL. 2019. PMDK man page: pmemobj_alloc. http://pmem.io/pmdk/manpages/linux/v1.5/libpmemobj/pmemobj_alloc.3

[16] INTEL. 2020. PMDK man page: libpmemobj - persistent memory transactional object store. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html.

[17] Intel Corporation. 2019. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/en-us/articles/intel-sdm.

[18] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://arxiv.org/abs/1903.05714v2

[19] Jaegeuk Kim. 2012. f2fs: introduce flash-friendly file system . https://lwn.net/Articles/518718/.

[20] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. 2020. Durable Transactional Memory Can Scale with Timestone. In *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Lausanne, Switzerland, 335–349.

[21] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A new file system for flash storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, 273–286.

[22] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. 2017. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Xi'an, China, 329–343.

[23] Ran Liu and Haibo Chen. 2012. SSMalloc: A Low-latency, Locality-conscious Memory Allocator with Stable Performance Scalability. In *Proceedings of the 3rd Asia-Pacific Workshop on Systems (APSys)*. Seoul, South Korea, 1–6.

[24] Micro. 2019. 3D XPoint Technology. https://www.micron.com/products/advanced-solutions/3d-xpoint-technology

[25] Gene Novark and Emery D Berger. 2010. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. Chicago, IL, 573–584.

[26] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 2017. Memory Management Techniques for Large-scale Persistent-main-memory Systems. In *Proceedings of the 43rd International Conference on Very Large Data Bases (VLDB)*. TU Munich, Germany, 1166–1177.

[27] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA, 241–254.

[28] Andrew T Phillips and Jack SE Tan. 2003. Exploring Security Vulnerabilities by Exploiting Buffer Overflow using the MIPS ISA. *ACM SIGCSE Bulletin* 35 (2003), 172–176. https://doi.org/10.1145/792548.611962

[29] Mustapha Refai. 2006. Exploiting a Buffer Overflow using Metasploit Framework. In *Proceedings of the 2006 International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*. New York, USA, 1–4.

[30] David Schwalb, Tim Berning, Martin Faust, Markus Dreseler, and Hasso Plattner. 2015. nvm malloc: Memory Allocation for NVRAM. In *Proceedings of the International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. Kohala Coast, HI, 61–72.

[31] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. Dallas, TX, 2389–2403.

[32] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: A Tunable Secure Allocator. In *Proceedings of the 27th USENIX Security Symposium (Security)*. Baltimore, MD, 117–133.

[33] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the 28th USENIX Security Symposium (Security)*. Santa Clara, CA, 1221–1238.

[34] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the 16th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, CA, 91–104.

[35] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *Proceedings of the 2018 ACM SIGMOD/PODS Conference*. Houston, TX, USA, 473–488.

[36] Matthew Wilcox. 2014. Add Support for NV-DIMMs to Ext4. https://lwn.net/Articles/613384/

[37] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, 323–338.

[38] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, 169–182.

[39] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. Renton, WA, 897–912.

[40] Pengfei Zuo and Yu Hua. 2018. SecPM: a Secure and Persistent Memory System for Non-volatile Memory. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*. Boston, MA.