POSEIDON: Safe, Fast and Scalable Persistent Memory Allocator

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

1 Introduction

Persistent Memory (PM) allocator is a critical component in Non-volatile Main Memory (NVMM) software stack managing program's NVMM space atop NVMM-aware file system (*e.g.*, ext4-DAX). It provides memory allocation/free from/to an NVMM heap similar to the traditional volatile memory allocator. Like volatile memory allocator (*e.g.*, jemalloc, tcmalloc), PM allocators should provide high performance and good multi-core scalability to effectively support a wide range of PM applications. However, the design of a PM allocator is fundamentally different than that of a DRAM allocator; especially in terms of the safety guarantees. Unlike the DRAM, NVMM is durable; so any corruption in the NVMM heap lasts forever and may render the underlying application irrecoverable. Therefore, a PM allocator must protect the NVMM heap metadata from a system crash as well as program bugs. The NVMM heap metadata corruption may cause serious consequences such as irrecoverable heap state, silent user data corruption and persistent memory leaks.

Although existing PM allocators [5, 7] rely on logging for heap metadata consistency, they are still vulnerable to heap metadata corruption due to program bugs. In particular, we found that a simple buffer overflow in a program can cause PMDK to reallocate the already allocated memory or fail to free the entire allocated space causing silent user-data corruption or persistent memory leak.

In this work, we aim to overcome the safety, scalability, and performance limitations of existing PM allocators and propose a novel persistent memory allocator, **POSEIDON**. To best of our knowledge, POSEIDON is the first PM allocator to guarantee complete heap metadata protection from not only crash but also program bugs and achieve high performance and scalability in tandem. In the rest, we will first discuss the problems in existing persistent memory allocator (§2). We then present our design (§3) and evaluation results (§4).

2 The Case of PMDK Memory Allocator

Heap Metadata Design. The PMDK persistent memory allocator, `libpmemobj` [5], manages its metadata in three different places. First, `libpmemobj` manages allocation bitmap and memory block headers on NVMM. They represent allocation state (*e.g.*, free or allocated, allocation size) and protected by undo logging. Next, for performance `libpmemobj` stores frequently accessed metadata on DRAM: arena structure and free-list for small-size allocation and a global AVL tree for large-size allocation. Lastly, it adopts in-place metadata layout storing allocation metadata (*e.g.*, allocation size) right before the allocated address for fast access. However, such metadata management in PMDK is vulnerable to program bugs and a centralized metadata (*e.g.*, the global AVL tree) often become a scalability bottleneck. Below we discuss the critical issues in PMDK allocator.

(1) In-place Metadata Corruption. When a program bug (*e.g.*, heap overflow) corrupts the in-place metadata, especially the size field, `libpmemobj` may free incorrect size of NVMM space. It may result in allocating the already-allocated space or failing to free the entire allocated memory. Duplicated allocation causes silent yet permanent user data corruption and failure in free causes permanent memory leak.

(2) Direct Metadata Corruption. Moreover, the allocator metadata on NVMM (*e.g.*, bitmap and memory block headers) is mapped to program's virtual address space with a read-writable permission. Hence, a software bug can corrupt the metadata directly resulting in an irrecoverable NVMM heap.

(3) Volatile Metadata Corruption. Similarly, metadata on DRAM (*e.g.*, free-list, the global AVL tree) is also in the same virtual address space with a read-writable permission. The DRAM metadata corruption results in an incorrect memory allocation/free which can corrupt NVMM heap metadata.

(4) Non-scalable Performance. We found two main bottleneck in `libpmemobj` upon concurrent allocation and free. First one is the global AVL tree used for managing large-size free blocks in NVMM can be the source of contention because it is protected by a global lock. In addition, when `libpmemobj` frees a small-size memory, it turns off a bit in the bitmap but does not immediately put the freed space to the free-list in DRAM. When the free-list becomes empty, `libpmemobj` re-build the free-list by re-scanning the metadata on NVMM. The allocation will be halt during the re-building process so it becomes a bottleneck in allocation/free-heavy programs.

3 Overview of POSEIDON Architecture

We design POSEIDON to provide complete heap metadata protection without any performance degradation. We illustrated the overall architecture of POSEIDON in Figure 1 and describe the key design features in the rest.

(1) Per-CPU Sub-Heaps. 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 lock for synchronization. Our per-CPU design not only reduces contention among multiple threads but also guarantees allocated memory is always NUMA-local. It allows multiple NVMM controllers on different NUMA domains to be used, fully utilizing NVMM

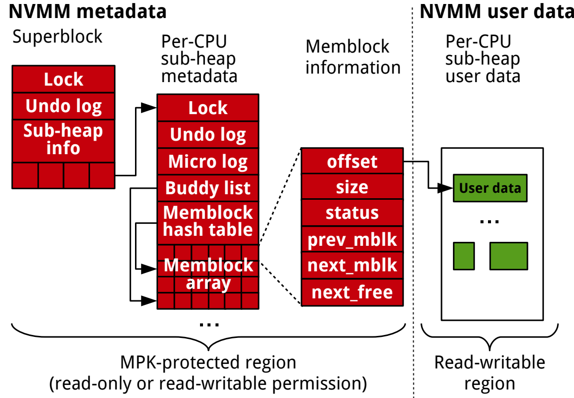

Figure 1: Heap layout of POSEIDON persistent memory allocator.

bandwidth.

(2) Fully Segregated Metadata Layout. POSEIDON completely segregates heap metadata region and user-data region. Superblock and per-CPU sub-heap metadata exist as a header, logically placed at the beginning of the POSEIDON persistent heap. The rest of the area is used for the 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 manage access permission in POSEIDON efficiently.

(3) Metadata Protection with Intel MPK. POSEIDON uses Intel Memory Protection Keys (MPK) [8] to efficiently change access permission of the metadata region for each thread. By default, a POSEIDON metadata region is not given write permission. Only during an allocation/free operation, POSEIDON temporarily grants write permission for the metadata region. Moreover, the MPK permission change is per-thread and fast taking around only 23 CPU cycles. Thus, the write permission is given solely to the thread executing the operation and reverts it back to read-only at the end of the operation. This design entirely prevents metadata corruption from program bugs, which has not been accomplished by any persistent memory allocator. Note that applying MPK protection to allocators using in-place metadata design (*e.g.*, PMDK allocator) is not feasible because MPK protection is per-page.

(4) Metadata Management Using Hash Table. POSEIDON uses a hash table (using an address as a key) to access memory block information in constant time. The hash table in POSEIDON also ensures the APIs cannot maliciously corrupt the metadata. Prior to any memory request (malloc/free), POSEIDON uses the hash table to validate the memory address and its status to prevent metadata corruption due to double-frees and invalid-free bugs.

(5) Crash Consistency. POSEIDON uses undo logging to ensure the failure-atomicity of the metadata update from the program/system crashes. In addition, POSEIDON uses micro

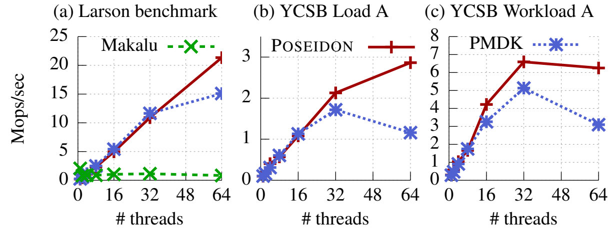

Figure 2: Performace of persistent memory allocator.

logging, which keeps the history of memory allocations to prevent memory leaks for transactional memory allocation.

4 Evaluation

We evaluate the performance of POSEIDON using a 2-socket server equipped with 768GB DRAM, and 3.0TB (12×256GB) of Intel Optane DC Persistent Memory (DCPMM). We use two benchmarks, YCSB benchmark and Larson benchmark.

Larson Benchmark. The Larson benchmark [2] simulates a server memory allocation pattern. 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 [7]. As Figure 2(a) shows, POSEIDON significantly outperforms other persistent memory allocators mainly because of per-CPU sub-heap design.

YCSB. The YCSB benchmark [3] simulates a key-value store scenario, performing multiple, concurrent, cross-thread allocations and deallocations in a persistent index structure as the workloads. We modified BZTree [1] to evaluate the performance of PM allocators as previous study [6] shows that BzTree suffers from PM allocation overhead. We ran experiments using two workloads, LOAD A (100% insert, allocation-heavy workloads) and Workload A (50% update and 50% read, allocation-less workloads). As Figure 2(b) shows, POSEIDON shows better multi-core scalability than PMDK in the allocation-heavy workloads due to its per-CPU sub heap design and hash table based metadata management. Also, POSEIDON shows better performance in the Workload A, as shown in Figure 2(c). Because POSEIDON can fully utilize the bandwidth of Intel Optane DCPMM in different sockets. For more information, refer to our full paper [4].

References

[1] Arulraj *et al.* Bztree: A High-performance Latch-free Range Index for Non-volatile Memory. VLDB 2018.

[2] Berger *et al.* Hoard: A Scalable Memory Allocator for Multithreaded Applications.ASPLOS 2000.

[3] Cooper *et al.* Benchmarking cloud serving systems with YCSB. SOCC 2010

[4] Demeri *et al.* Poseidon: Safe, fast and scalable persistent memory allocator. MIDDLEWARE 2020.

[5] pmdk-alloc URL `http://pmem.io/pmdk/manpages/linux/v1.5/libpmemobj/pmemobj_alloc.3`.

[6] Lersch *et al.* Evaluating persistent memory range indexes. VLDB 2020.

[7] Oukid *et al.* Memory Management Techniques for Large-scale Persistent-main-memory Systems. VLDB 2017.

[8] Park *et al.* Libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). USENIX ATC 2019