POSEIDON: A Safe and Scalable Persistent Memory Allocator

Anthony K Demeri

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Changwoo Min, Chair
William Diehl
Haibo Zeng

May 4, 2020
Blacksburg, Virginia

Keywords: Persistent Memory, Memory Allocator, Security, Scalability

POSEIDON: A Safe and Scalable Persistent Memory Allocator

Anthony K Demeri

(ABSTRACT)

With the advent of byte-addressable Non-Volatile Memory (NVMM), the need for a safe, scalable and high-performing memory allocator is inevitable. A slow memory allocator can bottleneck the entire application stack, while an unsecure memory allocator can render underlying systems and 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; critically, we also note: the de-facto NVMM allocator, Intel's Persistent Memory Development Kit (PMDK), is vulnerable to silent data corruption and persistent memory leaks as result of a simple heap overflow. We closely investigate the existing defacto NVMM memory allocators, especially PMDK, to study their vulnerability to metadata corruption and reasons for poor performance and scalability. We propose Poseidon, which is safe, fast and scalable. The premise of Poseidon revolves around providing a user application with per-CPU sub-heaps for scalability, 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 metadata safety.

POSEIDON: A Safe and Scalable Persistent Memory Allocator

Anthony K Demeri

(GENERAL AUDIENCE ABSTRACT)

Since the dawn of time, civilization has revolved around effective communication. From smoke signals to telegraphs and beyond, communication has continued to be a cornerstone of successful societies. Today, communication and collaboration occur, daily, on a global scale, such that even sub-second units of time are critical to successful societal operation. Naturally, many forms of modern communication revolve around our digital systems, such as personal computers, email servers, and social networking database applications. There is, thus, a never-ending surge of digital system development, constantly striving toward increased performance. For some time, increasing a system's dynamic random-access memory, or DRAM, has been able to provide performance gains; unfortunately, due to thermal and power constraints, such an increase is no longer feasible. Additionally, loss of power on a DRAM system causes bothersome loss of data, since the memory storage is volatile to power loss. Now, we are on the advent of an entirely new physical memory system, termed non-volatile main memory (NVMM), which has near identical performance properties to DRAM, but is operational in much larger quantities, thus allowing increased overall system speed. Alas, such a system also imposes additional requirements upon software developers; since, for NVMM, all memory updates are permanent, such that a failed update can cause persistent memory corruption. Regrettably, the existing software standard, led by Intel's Persistent Memory Development Kit (PMDK), is both unsecure (allowing for permanent memory corruption, with ease), low performance, and a bottleneck for multicore systems. Here, we present a secure, high performing solution, termed Poseidon, which harnesses the full potential of NVMM.

Dedication

This work is dedicated to my mother, Gayle, who has been an endless bounty of strength and resiliency for myself and others.

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Changwoo Min, who has put forth an endless channel of support for myself, the COSMOSS research team, and his many students, with unabated support. Dr. Min's resiliency in helping the team overcome technical, logistical, and personal obstacles has been of tremendous value to me; I am grateful for the opportunity I have been provided to study, learn, and grow under his guidance and leadership.

While working with Dr. Min, fortunately, I had the opportunity to share my time on the COSMOSS team with driven, intelligent, resourceful students and research assistants, specifically: Madhava Krishnan, Ajit Mathew, Jaeho Kim, Wookhee Kim, Mohannad Ismail, Xinwei Fu, Sumit Monga, and many others. These students worked with me, personally, from my first days in COSMOSS, as a rising Graduate student, to my final semester at Virginia Tech. I am forever grateful for their attention to detail, energy, time, and support on a technical and communal level.

Throughout my time as both an Undergraduate and Graduate student at Virginia Tech, the entire faculty, from the academic level to the logistical level, has been a part of my family of academia. It has truly been a pleasure to learn alongside the Virginia Tech faculty, who have gone above and beyond to provide support for myself and others to learn, grow, and succeed. Specifically, I would like to thank: Dr. A. Lynn Abbott, Mary Brewer, Dr. Tam Chantem, Dr. William Diehl, Dr. Wei Han, Dr. Michael Hsiao, Dr. Jia-Bin Huang, Dr. KJ Koh, Dr. Changwoo Min, Roxanne Paul, Dr. Paul Plassmann, Dr. TC Poon, Jason

Thweatt, Dr. Pratap Tokekar, Dr. Joseph Tront, Dr. Chris Wyatt, Sohei Yasuda, and Dr. Haibo Zeng.

Additionally, I would like to thank my former military leadership, who always encouraged myself and others to put forth our best efforts and strive for not only self-improvement, but, also the improvement of entities larger than ourselves, such as academia and family. Specifically, I would like to thank: Theodore Allen, Jose Mercado, Daryn Purcell, Steven Stalker, Yvan Uyen, Veronica Vila, Angelica Wallace, and Douglas Yoder.

Finally, I would like to thank my family, friends, and loved ones, for their continued support and, truthfully, incredible toleration of the many delayed phone call responses and postponed outings which were a necessary consequence of balancing my employment, health, and studies. Specifically, many thanks are due to: Gayle Cox, Tom Cox, Anthony Demeri, Grace Demeri, Alec Demeri, Alysha Demeri, Joe Green, Samuel Lam, Michael Mirda, and Lelia Cockerham.

Informally, thanks are undoubtedly due to those I have, in one way or another, crossed paths with along my time in academia; I am humbled and grateful to have shared such experiences.

Contents

List of Figures

Chapter 1

Introduction

Non-Volatile Main Memory (NVMM), such as Intel's Optane DC Persistent Memory [2, 29], is finally available to the public. It is expected to revolutionize storage systems and how we write programs by virtue of DRAM-like performance, byte-addressable persistence, higher density, cheaper price, and less energy consumption than DRAM.

Programs can now directly access NVMM without kernel intervention through mmap-ed NVMM memory by DAX-enabled file systems [11, 44, 47]. Programming with NVMM is conceptually similar with regular DRAM-based programming using rich, program-defined data structures and memory allocation. For example, Intel's Persistent Memory Development Kit (PMDK) [20] provides various container libraries for applications while additionally providing crash consistency, which guarantees consistency of NVMM data even after a crash (*e.g.*, program/system crash, sudden power outage).

In this direct, NVMM-access programming model, NVMM space management is divided into two parts: First, DAX-enabled file systems manage NVMM space of an NVMM device in a coarse-grained manner with files, and then persistent memory allocators manage NVMM space of a NVMM-backed file in a fine-grained manner. Persistent memory allocators [7, 20, 33, 38] provide memory allocation and free from/to an NVMM heap, similar to volatile memory allocators (*e.g.*, jemalloc [16] and TCMalloc [17]). In addition, most persistent memory allocators also provide transactional allocation and free of multiple NVMM objects, to support persistent transaction [20].

Persistent memory allocators have to meet two critical requirements:

1. Heap metadata safety
2. High performance and scalability

First, 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. Most existing persistent memory allocators [7, 33, 38], including Intel's PMDK allocator, rely on logging to avoid heap metadata corruption from a crash. Next, persistent memory allocators should provide high performance and scalability so they do not become a performance and scalability bottleneck for applications. For example, to reduce contention on accessing heap metadata from concurrent threads, PMDK allocator adopts a per-thread allocation structure on DRAM (called an "arena"), similar to scalable memory allocators for DRAM [4, 16, 17].

However, we found that state-of-the-art persistent allocators, such as PMDK [20] and Makalu [7], neither guarantee heap metadata safety nor provide scalability on true NVMM hardware. Additionally, we found that a simple programming mistake or a 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 (*i.e.*, overlapping allocation), causing silent user data corruption, or it can fail to find a free memory chunk, causing a permanent, persistent memory leak. *We argue that the current design of persistent memory allocators is fundamentally vulnerable to such a programming bug because the heap metadata either on NVMM or DRAM is mapped directly to the program's virtual address*

space. Also, even though persistent memory allocators, including PMDK and Makalu, are designed with multi-core scalability in mind, we found that they are not scalable in real NVMM hardware and can be a critical performance and scalability bottleneck in real-world applications. *We argue that coupling concurrency mitigation in allocator design, which has been commonly adopted in both the volatile and persistent memory allocators, is not a good fit for NVMM.* We will further discuss why existing persistent memory allocator designs fail to achieve heap metadata safety and scalability in the next chapter (Chapter 2).

To address the aforementioned problems, we propose Poseidon*, which is a persistent memory allocator designed for guaranteed heap metadata safety and multi-core scalability.* More specifically, we made following contributions in this thesis:

- We found that the state-of-the-art persistent memory allocator provided neither guaranteed heap metadata safety nor multi-core scalability. In particular, we demonstrated 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.

- We introduce a new persistent memory allocator, named Poseidon. Our main design goal is to guarantee heap metadata protection and achieve high performance and scalability in tandem. To this end, we propose a new heap metadata layout, which is completely segregated from user data and efficiently protected by a hardware memory protection feature (Intel Memory Protection Keys [13, 34]). Also, we propose a per-CPU sub-heap layout, which effectively reduces contention from concurrent threads and fully utilizes hardware resources (especially persistent memory controllers) in multi-socket systems.

- We implemented Poseidon and evaluated Poseidon for micro-benchmarks and real-world applications. Our experimental results show that applications using Poseidon

outperform up to 242 times when compared to other state-of-the-art persistent memory allocators.

The rest of the thesis is organized as follows. Chapter 2 provides the motivation for Poseidon's approach, with a case study for PMDK. Chapter 3 describes Poseidon's design and Chapter 4 shows Poseidon's implementation. Chapter 5 shows our evaluation results with micro-benchmarks and real-world applications. Chapter 6 discusses the limitation and potential future directions of Poseidon and provides suggestions for PMDK. Chapter 7 compares Poseidon with previous research and Chapter 8 concludes.

Chapter 2

Background and Motivation

Emerging NVMM (also known as persistent memory or PM) has similar characteristics to DRAM [12, 25, 41]. It is byte-addressable and has DRAM-like high performance. Thus, many research efforts aim to use NVMM as a memory heap. In this approach, the role of a persistent memory allocator becomes critical [6, 9, 20, 25, 28, 30, 32, 33, 37, 43, 46, 50].

In this chapter, we describe the characteristics of and requirements for persistent memory allocators which manage NVMM. Also, we describe our analysis on Intel's PMDK persistent memory allocator, libpmemobj, which is the state-of-art, open-sourced persistent memory allocator of today.

2.1 Persistent Memory Allocators

Persistent memory allocators manage memory pools as mmaped persistent memory spaces. Since persistent memory is byte-addressable, memory management techniques are quite similar to that of DRAM. However, unlike DRAM, persistent memory allocators [6, 20, 33, 37] typically adopt 16-byte pointers, which store a memory pool id and offset in a logical pointer. Since most applications are designed for 8-byte pointer addresses, this slightly increases the complexity of using persistent memory allocators. Moreover, since the persistent memory allocator manages memory space as a pool, additional metadata (such as locks and index structures) for managing memory pools is also required.

In addition to 16-byte pointers, NVMM also requires additional metadata management efforts because of its non-volatility, specifically: 1) root pointers, 2) crash consistency adherence [19, 22, 45, 49]. A root pointer points to a known space in a memory pool; it guarantees one can always find a specific, static location in a memory pool. 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.

Traditional transient memory allocators do not consider crash consistency, such as sudden power outage, because DRAM is volatile. If a sudden power outage occurs, data in DRAM will be lost, and heap metadata on DRAM will be reconstructed after the system reboots.

However, for persistent memory allocators, it is essential to guarantee the crash consistency of heap metadata, which contains critical metadata, such as allocation bitmaps and free memory space management information.

To guarantee crash consistency of heap metadata, most persistent memory allocators use a logging approach. There are two kinds of logging used for crash consistency in NVMM: 1) *undo*, 2) *micro*. An *undo-log* is used to guarantee a consistent state of metadata for persistent memory allocators. A *micro-log* is used to prevent persistent memory leaks by keeping track of memory allocation records.

Support for transactional allocations is an additional requirement for persistent memory allocators to guarantee crash consistency. Since NVMM is non-volatile, critical metadata errors cannot be resolved simply by restarting the application or system [3, 8, 9, 12, 14, 19, 22, 25, 28, 30, 32, 48, 49, 51]. For example, when an application tries to allocate an NVMM space to store data, an allocator allocates memory space for the application in the NVMM, updating the metadata for the allocated space before returning the address of newly allocated space to the application. If a system failure occurs after the metadata

update, but before returning to the application, the NVMM space will be lost permanently (*persistent memory leak*). Similarly, when an application tries to deallocate NVMM space, the persistent memory allocator updates its metadata first and then returns the result to the application. So, if a system failure occurs between metadata update and returning to the application, the application doesn't know whether the space is deallocated or not; the application may thus try to deallocate space which is already deallocated (*double free*). Transactional allocations can prevent these critical errors (*persistent memory leak, double free*) because the transaction is completed atomically. When system failure occurs during a transactional allocation, it recovers the metadata of the heap and deallocates space allocated in the transactional allocation by using internal recovery protocol (micro-logging) [20, 33].

Safety is another critical issue for persistent memory allocators. The metadata of persistent memory allocators contains important information regarding the allocation status of NVMM space. The characteristics of persistent memory allocator metadata are quite similar to those of file systems. In file systems, the consistency of metadata is guaranteed by journaling [1, 10]; metadata is additionally managed in the kernel space. The metadata is thus safe from user-program bugs due to the isolation between kernel and user space. Now, since persistent memory allocators [6, 20, 33] guarantee crash consistency of the metadata using a logging approach, and since persistent memory allocators are implemented in user-space, there is no isolation between user data and metadata of persistent memory allocator. Additionally, commonly-used, in-place metadata, stored in user space, makes it much more difficult to protect metadata from corruption, based on its close proximity to user data and inability to be distinguished from user-data on the page-level.

While providing safety, persistent memory allocators should also maintain high performance and manycore scalability. Since every memory access request is managed by persistent memory allocators, it must not be a bottleneck on application performance. In addition,

since commercially available NVMM provides much larger capacity than that of DRAM. It means that persistent memory allocators should be scalable in both core and memory capacity.

2.2 The Case of PMDK

libpmemobj uses both NVMM and DRAM to maximize the performance advantages from DRAM. Since the latency of DRAM is five times faster than that of NVMM [23], *libpmemobj* stores frequently accessed data structures in DRAM, as shown in Figure 2.1.

libpmemobj stores arena structures, free-lists, and an AVL-Tree in DRAM. The arena structures, free-lists, and AVL-Tree 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 considers multi-core systems by using a per-thread structure, defined as an arena. In this structure, libpmemobj stores its metadata and allocated objects. The NVMM space is further divided into zones; each zone consists of chunks. Plainly, libpmemobj manages objects by their sizes. If an allocated object size is small, the object is stored in a chunk and managed with an allocation bitmap and memory object header, which includes metadata of the allocated object. It stores metadata of the allocated object as in-place metadata to the user-space that object is allocated. In-place metadata is used to improving performance; moreover, since the metadata is located in a nearby space, libpmemobj can easily find metadata from a cache-line neighbor. When an allocated object size is bigger than a given threshold, it is stored as a large chunk with a memory block header (instead of an allocation bitmap). To guarantee the data consistency of the metadata, libpmemobj uses undo logging.

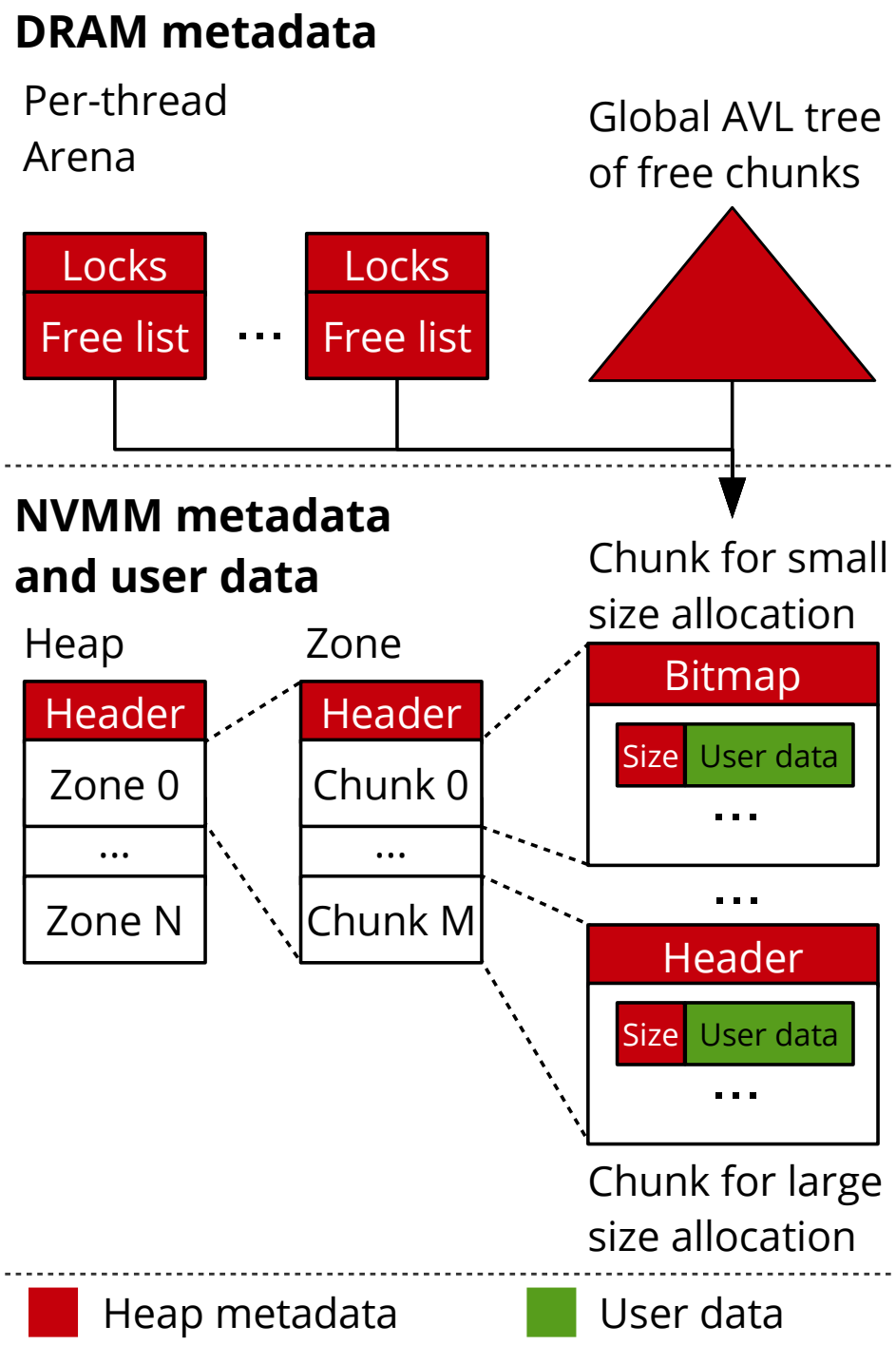

Figure 2.1: Architecture of PMDK persistent memory allocator.

Overall, the techniques used in libpmemobj are focused on improving performance (reduced latency). For example, they exploit DRAM as cache layer and use in-place metadata to

```
1  void pmdk_overlapping_allocation(nvm_heap *heap) {
2      void *p[1024], *free;
3      int i;
4
5      /* Make the NVM 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 NVM 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 NVM 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 NVM 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 2.2: PMDK heap metadata corruption caused by heap overwrite. The corruption of in-place metadata in Lines 16, 46 can cause overlapping allocations (pmdk_overlapping_allocation) and permanent memory leaks (pmdk_permanent_leak). For brevity, we use the traditional malloc/free-like API instead of using PMDK's memory allocation APIs.

improve the cache locality.

However, those techniques make libpmemobj more vulnerable to user application bugs. Since libpmemobj works as a user-level library, there is no isolation between metadata and user data; even worse, in-place metadata is stored near user data, so, if a user application has a memory bug which modifies in-place metadata, the metadata will be corrupted [15, 35, 36, 49, 51]. We further discuss this metadata corruption in the next chapter.

2.2.1 Heap Metadata Corruption from Application Bugs

The importance of persistent memory allocator metadata cannot be overstated; it keeps the record of allocated data and free space in a persistent memory heap. Once this metadata is corrupted, a memory allocator will lose allocated data and free space management information which can cause a permanent memory problem (a memory leak or over-allocation problem, for example). Our study of libpmemobj found out that there are three routes for heap metadata corruption 1) Direct metadata corruption, 2) Indirect metadata corruption through in-place metadata in a user allocated space, and 3) Indirect metadata corruption through volatile cached metadata in persistent memory heap. Let us explain in more detail.

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 user application modifies the bitmap directly, it can lose stored objects.

Indirect metadata corruption through in-place metadata in user allocated space. In libpmemobj, the metadata of an allocated object is stored right before the allocated object as an object header. If libpmemobj deallocates an object from the persistent memory heap, first, it checks the size of the object from the object header. In the case of small-sized memory allocations, it unsets the bit corresponds to the size in the object header. As shown in Figure 2.2, Line 16, when an application has a bug, it modifies the object header's size value to larger than object size, libpmemobj unsets the allocation bit in the bitmap by the object header's size. Hence, libpmemobj adds space to the free space such that it allocates new objects to overlapped space. Another example is when an application modifies an object header's size to a smaller value than the object's size, as shown in Figure Figure 2.2, Line 46;

here, the deallocation process of libpmemobj reclaims only the smaller size. In this case, we cannot reclaim the remaining space, permanently, causing a permanent persistent memory leak. It is more problematic in real-world applications because misuse-of-pointer bugs can easily occur overwrites.

Indirect metadata corruption through volatile cached metadata in PM heaps. Since libpmemobj uses both free-lists and an AVL-tree in DRAM, if the information in DRAM is not correct, memory allocation/deallocation can corrupt persistent memory heap. To prevent this problem, we need to protect both DRAM layers and NVMM layers.

2.2.2 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 data structures in DRAM. For example, 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. Eventually, there is no free space in the free-list, so it reads a bitmap of chunk-size and rebuilds the free-list. With this design, it is able to minimize the deallocation overhead and protect misuse or malicious use of API such as double free and invalid free. For example, suppose libpmemobj deallocates small objects in NVMM and puts the deallocated space in the free space instead of unsetting the bitmap. In this case, when an application deallocates same NVMM space twice, it would put the same deallocated space to free-list twice, allowing allocated objects to be easily corrupted.

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. Although libpmemobj

uses per-thread arena structures for better scalability, the global AVL-tree becomes a source of contention and hurts scalability of libpmemobj for large allocations.

Finally, we find a major bottleneck of libpmemobj performance is its failure to utilize hardware resources effectively. Specifically, we know a given hardware has a limited number of memory controllers per socket; these sockets are also, naturally, connected by a socket interconnect (which has a performance-limiting bandwidth). Now, by design, libpmemobj creates all sub-heaps on a single socket; thus, as the number of utilized cores (and, tangentially, the number of utilized sockets) increases, libpmemobj necessarily fails to utilize the maximum number of per-socket memory-controllers (a fundamental component of persistent memory storage), while over-utilizing the socket interconnect, introducing significant performance bottlenecks. Truly, these significant bottlenecks are a direct consequence of the design of libpmemobj; the removal of such a bottleneck would be non-trivial for PMDK design.

Chapter 3

Design of Poseidon

We first describe our design goal (section 3.1), design overview (section 3.2), and then discuss the detail design (section 3.3) of Poseidon.

3.1 Goals

We aim to design a persistent memory allocator, Poseidon, to fulfill 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 API (*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 API as shown in section 2.2. 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

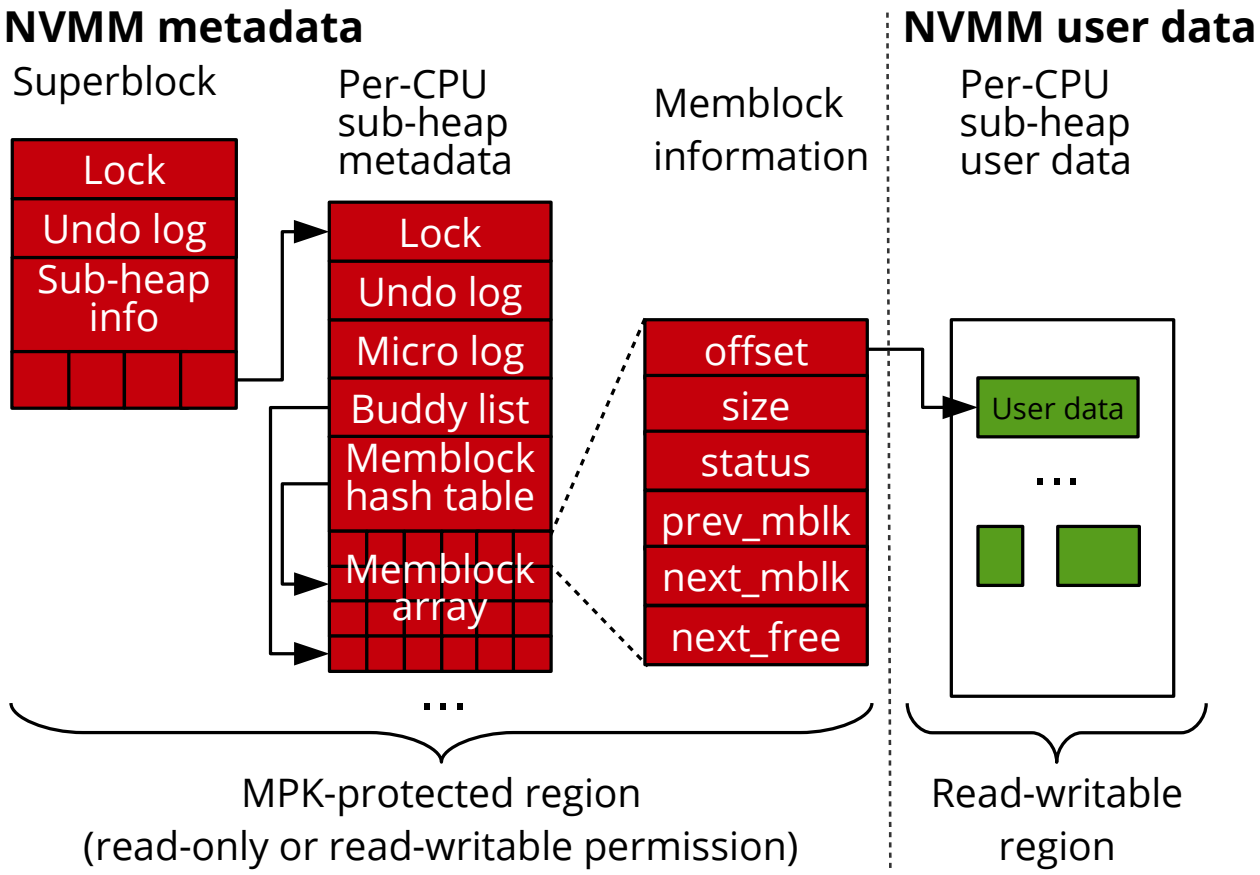

Figure 3.1: Heap layout of Poseidon persistent memory allocator

fundamental requirement for all persistent memory allocators.

3.2 Overview

We illustrate the design of Poseidon in Figure 3.1. Poseidon completely segregates heap metadata from user data. The entire heap metadata is protected by an efficient hardware protection mechanism, Intel's MPK, so it becomes read-writable only when running Poseidon code for a given thread. This guarantees complete heap metadata safety. In addition, crash consistency is guaranteed by using undo logs for singleton allocations and micro log for a transactional allocations. The Poseidon heap consists of per-CPU sub-heaps to minimize

synchronization costs of concurrent malloc/free operations, while guaranteeing NUMA-local allocations, in order to fully utilize scarce hardware resources (especially memory controllers). Poseidon maintains information of all allocated/free memory blocks to prevent double-free and invalid-free bugs. Memory blocks are managed by an extensible, multi-layer hash table, so the cost of Poseidon operation is constant as NVMM heap increases. In the rest of this chapter, we describe key features of Poseidon and explain how Poseidon fulfills the design goals.

3.2.1 Per-CPU Sub-Heaps

The Poseidon heap consists of per-CPU sub-heaps, which have two advantages over typical allocator design. First, a per-CPU design guarantees allocated memory is always NUMA local, so applications do not need to 'pay' remote NVMM costs across the NUMA domain in the common case. More importantly, it allows NVMM memory to be physically spread among multiple persistent memory units. This allows multiple NVMM controllers (which are per-NUMA in x86 architecture) to be used, fully utilizing valuable hardware bandwidth.

3.2.2 Fully Segregated Metadata

We propose managing NVMM heap metadata as fully segregated metadata, as shown in Figure 3.1. Specifically, Poseidon maintains a single *superblock* for managing the metadata of all *per-CPU sub-heaps*. Within each *sub-heap* exists all metadata which is required for providing: allocations, deallocations, and metadata protection. Specifically, each sub-heap maintains its own: *buddy list* of memory blocks, *hash table* of memory blocks, *metadata lock*, *undo log*, and *micro log*. All of this metadata exists as a logical header to the sub-heap region of *user data*; in other words, there is *no in-place metadata* unlike other state-of-the-art persis-

tent allocators (*e.g.*, PMDK). 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.

3.2.3 Metadata Protection with Intel MPK

Poseidon separately manages the permission of metadata regions from the permission of user data regions. By default, a Poseidon metadata region is not given a write permissions. Only during a malloc/free operation will Poseidon temporarily grants write permissions (*i.e.*, read-writable) for the metadata region to solely the thread executing said malloc/free operation. This design entirely prevents metadata corruption due to software bugs, which is currently not accomplished by any persistent memory allocator [7, 20, 33, 38]. To efficiently change access permissions of a metadata region, Poseidon uses Intel Memory Protection Keys (MPK) [21, 34]. MPK is a new hardware feature which allows user-space applications to efficiently change memory access permissions; notably, since MPK arrived prior to NVMM, all hardware which supports NVMM also supports MPK. Now, MPK allows a contiguous region of memory to be assigned to a protection key, using a similar API to an mprotect system call. Once a contiguous region of memory is mapped to a given protection key, changing the memory access permissions of the region is both scalable and low latency–it only takes around 23 CPU cycles, regardless of the size of a memory region and the number of concurrent threads, using a non-privileged instruction wrpkru. 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. Note that Poseidon can thus use one MPK protection key for all sub-heaps.

3.2.4 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, while tree structures (*e.g.*, AVL tree [20] or NVMM-optimized B-tree [33]) have been commonly used to manage such information in memory allocators. The main advantages of using a hash table is the maintenance of constant time allocation and free operations, regardless of both the allocation size and the heap size. To dynamically re-size the hash table, Poseidon uses a multi-level hash table, similar to one used in F2FS file system [24, 26]. The hash table in Poseidon also ensures the API cannot be maliciously used to corrupt metadata with a hash table. 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 corrupt metadata.

3.2.5 Crash Consistency

The main difference between DRAM memory allocators and persistent memory allocators introduces the requirement for guaranteeing crash consistency. Guaranteeing crash consistency in persistent memory allocators prevents an inconsistent state of NVMM heap metadata. Poseidon uses a combination of *undo logging* and *micro-logging* to provide protection against system failure. By using undo logging, Poseidon writes the original, unmodified

```
1  // Initialize a Poseidon heap with a given size and path name
2  heap_t *poseidon_init(const char *heap_path, size_t heap_size);
3  // Deinitialize a Poseidon heap
4  void poseidon_finish(heap_t *heap);
5
6  // Allocate an NVMM space with a requested size
7  nvmptr_t poseidon_alloc(heap_t *heap, size_t sz);
8  // Transactionally allocate a memory with a flag is_end
9  // denoting whether this is the last allocation in a transaction
10 nvmptr_t poseidon_tx_alloc(heap_t *heap, size_t sz, bool is_end);
11 // Deallocate an NVMM space pointed by ptr
12 void poseidon_free(heap_t *heap, nvmptr_t ptr);
13
14 // Convert a NVMM pointer to a raw pointer
15 void *poseidon_get_rawptr(nvmptr_t ptr);
16 // Convert a raw pointer to an NVMM pointer
17 nvmptr_t poseidon_get_nvmptr(void *p);
18
19 // Get the pointer of a root object
20 nvmptr_t poseidon_get_root(heap_t *heap);
21 // Set the pointer of a root object
22 void poseidon_set_root(heap_t *heap, nvmptr_t ptr);
```

Figure 3.2: Poseidon API

metadata to the undo log prior to updating metadata. With undo logging, Poseidon ensures partially written metadata can be restored to its original state, thus preventing metadata corruption due to a program/system crash or sudden power outage. Additionally, Poseidon records the history of memory allocations (*i.e.*, an offset of successfully allocated NVMM memory) in a transactional allocation by using micro-logging. With micro-logging, Poseidon can prevent persistent memory leaks by allowing multiple memory allocations to be managed as a single transaction. Thus, when a transactional allocation aborts due to system failure, Poseidon deallocates memory addresses in the micro log, so the aborted transaction does not permit any persistent memory leak. While transactional memory allocation is, naturally, essential, many existing memory allocators [7, 33] overlook this requirement.

3.2.6 Programming Interface

Poseidon also provides a simple and intuitive programming API as shown in Figure 3.2 where most APIs are self-explanatory. A program first need to initiate (or create) a Poseidon heap

using poseidon_init, which returns a pointer to Poseidon heap. 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 regular, 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.

The representation of a persistent pointer in Poseidon similar to other persistent memory allocator. Poseidon's persistent pointer type nvmptr_t holds 8-byte heap ID, 2-byte sub-heap ID, and 6-byte offset to the sub-heap. Poseidon provides pointer conversion APIs between a raw pointer and a Poseidon persistent pointer (poseidon_get_rawptr, poseidon_get_nvmptr). 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.

3.3 Detail Design

3.3.1 Loading the NVM Heap

Upon initialization of Poseidon, an NVMM heap is loaded, which encompasses: an MPK protection key allocation for metadata protection, the recovery of inconsistent metadata (crash recovery), and the unlocking all metadata locks. Since the superblock is persistent, upon initialization, Poseidon interfaces with the persistent superblock metadata, immediately using its own internal undo log to ensure its internal state is consistent. Next, since the state is guaranteed to be consistent, Poseidon iteratively maps all existing sub-heaps to persis-

tent memory and maintains their sub-heap pointers locally. Since both the superblock and sub-heaps are now mapped to memory, Poseidon ensures this memory region is protected by allocating an appropriate MPK key and protecting all metadata regions from external write access. Finally, Poseidon ensures each sub-heap is in a consistent state by processing their respective undo logs and micro logs, which are only non-empty if a system failure occurred. Now, since Poseidon has appropriately recovered both the superblock and all sub-heaps, all metadata locks are unlocked and the NVM heap has been successfully loaded.

3.3.2 Reduced Metadata Footprint

In any system, reducing the underlying storage footprint is ideal. In particular, for a persistent memory allocator, which must persistently retain critical metadata for a potentially large number of allocations, a low storage footprint is arguably a hard-requirement for a high performaning product. In Poseidon, since our design uses binning metadata, which always enforces page-aligned metadata sectors, we are able to capitalize upon the fallocate syscall to rapidly modify the underlying file system block state (from ”in use” to ”free”) for any metadata page. Notably, since PMDK [20] uses in-place metadata, which cannot be delineated on a page level, PMDK can not use this technique without significant, non-trivial design changes. In this model, specifically, Poseidon reduces its metadata footprint by *hole-punching* unused metadata regions, returning them to the underlying filesystem, while still allowing the virtual address to be immediately writable, when needed. Poseidon primarily uses this technique to eliminate unused levels of its *multi-level hash table*. For example, upon sub-heap initialization, Poseidon will obtain required storage for 65M allocations, using a primary hash table with 1M allocations, and 64 multi-level hash tables, each allowing 1M allocations. Next, Poseidon will immediately ”hole-punch” (using the fallocate syscall) the storage for all multi-level hash tables, returning *536MB* of storage to the filesystem, reducing

metadata overhead for hash tables by *98%*. Now, when Poseidon needs to access a "hole-punched" region of memory, since it has already been mapped, Poseidon need only write to its known address; the memory is, in other words, immediately usable. Of course, based on the above implementation, it would be possible to eventually write to all of the previously hole-punched pages of memory, once again occupying space on the underlying filesystem; to mitigate this, Poseidon periodically *reclaims and returns* unused hash tables to the file system, so the hash table memory is only utilized when needed. To our knowledge, no other persistent memory allocator currently utilizes such a technique for minimizing system metadata footprint; in fact, since many persistent memory allocators utilize in-place metadata, such as Makalu, PMDK, and Pallocator [7, 20, 33], existing persistent memory allocators cannot trivially be adapted to utilize such a technique.

3.3.3 Allocation

Persistent memory allocations fall into one of three categories: 1) standard allocation, 2) defragmentation, or 3) transactional allocation. Let us explain in detail one by one.

Standard Allocation. When a user requests a standard allocation, Poseidon begins by changing the permissions of heap metadata to read-writable using MPK. Poseidon manages free memory blocks using a buddy list, which is an array of free lists where each list contains only a certain size of free blocks. Based on the requested allocation size, Poseidon accesses the sub-heap's internal buddy list and either: obtains an existing free block, splits larger free blocks until a free block is available, performs defragmentation or returns NULL. Poseidon chooses a sub-heap where the requesting thread is running. If a large enough free block is found, its memory block status is updated in the hash table, which is indexed using an O(1) hash algorithm. If a collision is detected during hash table indexing, first, linear

probing is used. Now, 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). Finally, the hash table has been appropriately updated with the allocation metadata (which is updated using the sub-heap's internal undo log). Thus, when a user has completed their internal storage of persistent metadata, the undo log transaction can be committed to persistent memory by atomically truncating the undo log and changing the metadata protection to read-only via MPK.

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 right 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 both scalable and high performance, contrary to the design of existing persistent memory allocators, which perform defragmentation on a global size-independent level [33] or a global level [7].

Transactional Allocation. In a transactional allocation, for each sub-allocation, all internal metadata is updated in an identical manner to a non-transactional allocations, with the addition of persisting the allocated address in an internal micro-log. Thus, a user can perform multiple sub-allocations within a given transaction, all of which can either be recovered or released following a system failure. At the conclusion of a given transaction, once a user has effectively stored the persistent offsets for their respective memory allocations, the user can commit the transaction, which truncates the micro-log in a single, atomic operation.

3.3.4 Deallocation

Memory deallocations from a sub-heap follows a near-inverse process to that of memory allocations. First, Poseidon grants the write permission to the metadata using MPK. Then, based on the user's proposed memory address to free, Poseidon searches the hash table on an appropriate sub-heap, using the address as a key. If the address is not found, 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. Otherwise, the memory block status is changed to free and inserted to the tail end of its respective size class of the buddy list, to prevent immediate reuse of the allocation. Finally, after all metadata updates have successfully been made to the respective sub-heap metadata and undo-log, the deallocation is committed by truncating the undo log and changing the sub-heap metadata permissions to read-only using MPK.

3.3.5 Crash Consistency and Recovery

Since all persistent memory allocators and applications must be robust to system/application failure, there must be an effective means of recovery; in Poseidon, we have noted our use of both undo logging and micro logging to accomplish this robustness.

Upon Poseidon initialization, all undo logs and micro logs are checked for data; since a successful transaction results in truncation of both undo logs and micro logs, the presence of data within either logs indicates a crash occurred.

Poseidon first checks if an undo log is non-empty. If so, it 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 which, the contents of the undo log are copied to the original metadata location and persisted to storage. Conclusively, after the undo log has been processed, the log is truncated

and the metadata protections are once again set to read-only. This process reverts the heap metadata to the last consistent status.

Then Poseidon checks if a micro log is not empty. If so, it means a crash occurred during a multi-allocation transaction. As such, all addresses within the micro log are deallocated prior to truncating the micro log and completing recovery. This process aborts an uncommitted transaction.

Chapter 4

Implementation

We implement Poseidon in C, comprising of 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. In order to, from a software standpoint, guarantee the persistence of data in NVMM, we use the clwb instruction (with sfence), which efficiently flushes CPU cache lines to physical NVMM.

Chapter 5

Evaluation

In this chapter, we first discuss the metadata safety guarantee of Poseidon (section 5.2). We then evaluate the scalability of Poseidon compared to PMDK [20] and Makalu [7] (section 5.3). We finally show the performance of real-world applications with Poseidon, in order to demonstrate the impact of Poseidon (section 5.4).

5.1 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 (2.70 GHz), 768GB DRAM, and 3.0TB (12 x 256GB) of Intel Optane DC Persistent Memory (DCPMM). 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 [33] because its source code is not publicly available.

5.2 Metadata Safety Evaluation

To guarantee metadata safety of a persistent memory allocator, three critical conditions must be considered:

- Is the metadata safe from non-deterministic systems or program failures?
- Is the metadata safe from arbitrary program bugs?
- Is the metadata safe from misuse of the APIs?

First, to protect metadata from system or program failures, Poseidon uses undo logging for writes to persistent memory, incorporating an undo logging mechanism within each sub-heap to maintain scalability. In addition, Poseidon uses micro logging for transactional allocation.

Next, we find both PMDK and Makalu do not adequately protect persistent metadata from corruption, as demonstrated in Figure 2.2; in fact, by over- or under-writing the in-place metadata surrounding memory returned from an allocation, the metadata can be persistently corrupted, causing persistent memory leaks and allowing memory to be over-allocated. To completely prevent the metadata corruption from program bugs, Poseidon adopts a fully segregated metadata layout. Furthermore, Poseidon grants write permissions for the metadata region only internally, and only to the presently running thread. We also manage to eliminate memory protection overhead and cross-thread privilege issues by adopting Intel MPK [34]. By definition, Intel's MPK manages privileges on a per-thread basis; when one thread's permission to a given region of memory change, other threads permissions remain unchanged, necessarily. In addition, MPK incurs near-zero overhead when changing memory permissions (23 * 2 CPU cycles) [34], which thus adds minimal latency to the critical path, allowing safe and scalable metadata protection.

Finally, to ensure prevention of malicious API usage, we prevent both double frees and invalid frees from occurring, so as to prevent metadata corruption. Poseidon checks if the requested address and its status are correct upon free. We do this in constant time by using a multi-layer hash table to manage memory blocks, rather than using tree structures [20, 33]; thus, regardless of the pool size or allocation size, allocation and free time is constant, which,

as observed in the evaluations for both PMDK and Makalu, is not always true.

5.3 Scalability for Multi-threading

We present three groups of benchmarks, aimed at demonstrating the scalability and performance of Poseidon via both microbenchmarks and real-world benchmarks. These benchmarks, listed below, demonstrate: not only does Poseidon provide metadata protection not offered by PMDK or Makalu, but Poseidon also provides better performance and scalability to many cores.

5.3.1 Allocation and Deallocation Microbenchmark

We first ran a mircobenchmark that performs 100 allocations and 100 frees in a random order, repeating this process such that a total of 1 million allocations/deallocations, with varying numbers of threads and allocation sizes, are performed.

As Figure 5.1 shows, Poseidon significantly outperforms all persistent memory allocators by upto *60x* and shows near linear scalability upto 100 threads. Poseidon is able to maintain scalability by using per-CPU sub-heap contained metadata. Moreover, constant time for memory allocation/deallocation 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 in low latency. So Poseidon shows better performance and scalability against other persistent memory allocator nearly for all contention levels, while providing safety.

Makalu [7] 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

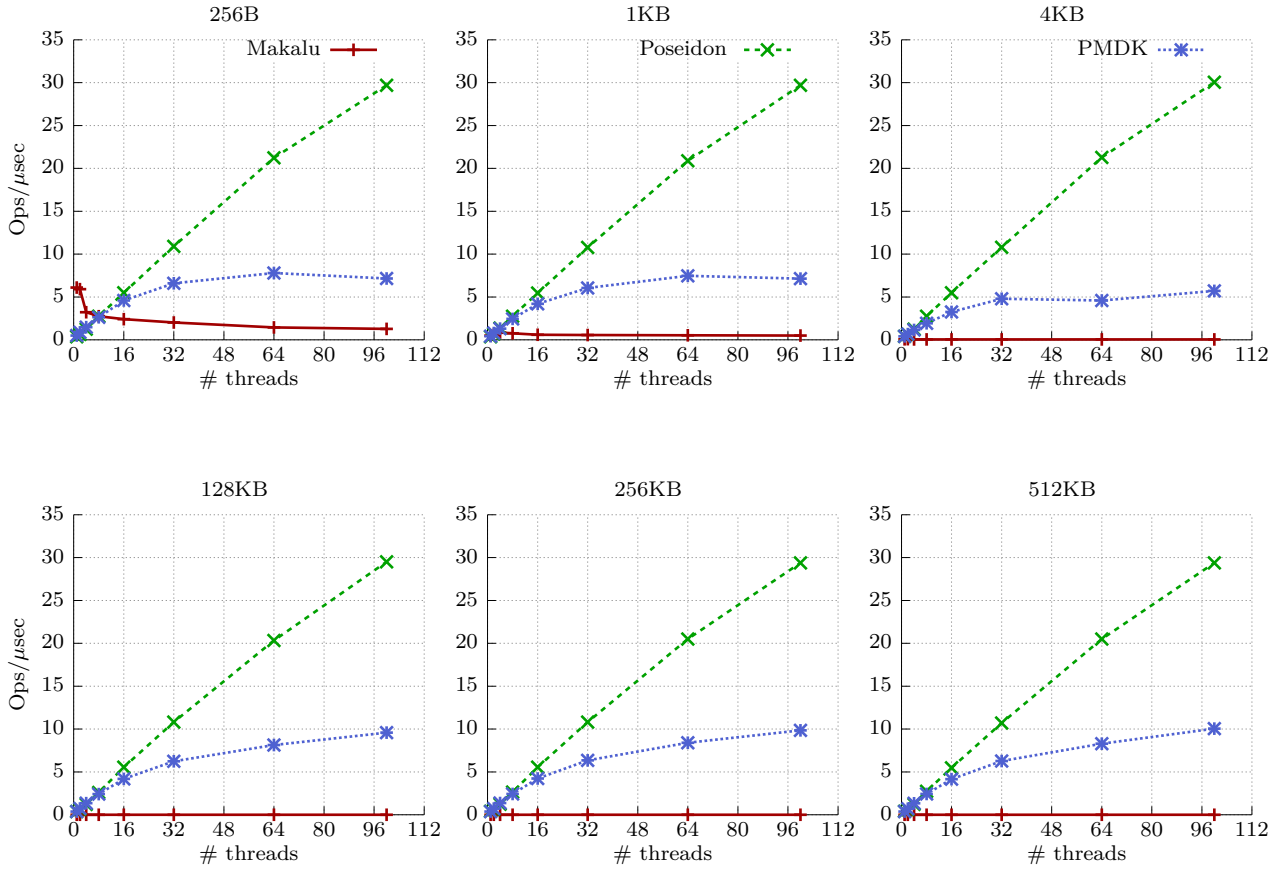

Figure 5.1: Performance of a pair of 100 mallocs and 100 frees in random order with different allocation sizes

incorporates a *global chunk list* for allocations greater than 400 bytes. Thus, 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 5.1, with a block size less than 400 bytes, we observe a 6x perfor-

mance loss, but, with a block size greater than 400 bytes, we observe greater than a 1000x performance loss.

The de-facto standard persistent memory allocator PMDK [20] shows better scalability than Makalu. However, it also saturates the performance when the number of threads are 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 section 2.2. 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; unfortunately, for free-heavy applications, this action log becomes a system bottleneck, clearly observed in Figure 5.1, where PMDK exhibits inverse performance as the number of threads surpasses 16.

5.3.2 Larson benchmark

The Larson benchmark [4] simulates a server: performing multiple, concurrent, cross-thread allocations and deallocations; in the persistent allocator technical community, this is the standard benchmark for comparison. This benchmark generates N objects of allocation/deallocation while varying the objects size randomly. We run the Larson benchmark for 10 seconds with a varying number of threads, as is the community standard. Similar to the results of our microbenchmarks, Poseidon significantly outperforms other persistent memory allocators upto *4x*. Once again, this is due to the bottlenecks which are observed and noted in

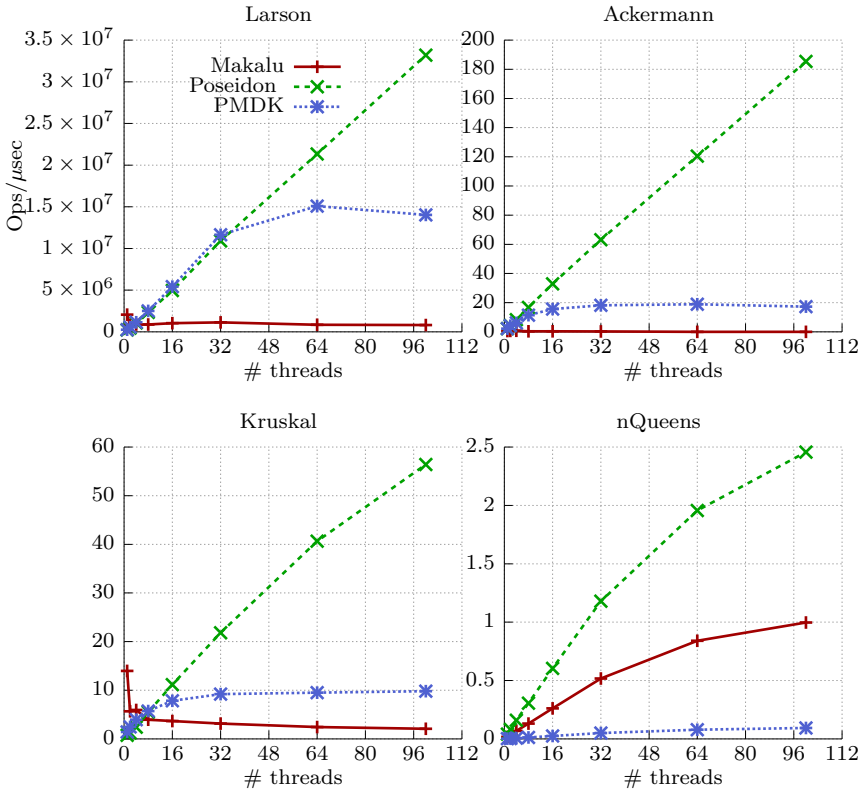

Figure 5.2: Performance of Larson benchmark and real-world benchmark

subsection 5.3.1.

5.4 Real World Application

Finally, to further demonstrate the performance and scalability of Poseidon, we provide real world examples of allocator use in computation-intensive applications. Notably, since the allocated memory regions in these benchmarks are heavily read-from and written-to, PMDK's failure to utilize valuable hardware memory controllers and failure to minimize usage of the socket interconnect, due to their non per-CPU heap design, yields a significant performance bottleneck when running high-thread benchmarks, as shown in the results. In these high performance benchmarks, we evaluate the effectiveness of 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 from 1 to 100.

Ackermann benchmark. We perform a single, 1GB allocation, which is filled in with Ackermann results upto (4, 5); then, this region is allocated, utilized as cache to compute Ackermann results, deallocated, and repeated 100,000 times. Poseidon performs upto *242x* better than Makalu and *10x* 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 8, since the computation intensive nature of the benchmark exceeds reclaim list contention at this point; 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 upto *5x*, *28x*, 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 is complete; similar to the Ackermann and Kruskal benchmarks, this process is repeated 100,000 times. The performance of Poseidon is *2.4x* and *24x* better than Makalu and PMDK respectively. Interestingly, Makalu shows better performance than that of PMDK, which is due to Makalu's per-thread, delayed mapping of memory (which maps the memory closer to the running-thread's socket), as opposed to PMDK's requirement to create pools in the main thread (which maps memory potentially farther from a running-thread's socket, thus failing to utilize the maximum

number of memory controllers, and increasing socket interconnect contention).

We also note an additional benefit provided by Poseidon. Since Poseidon sub-heap creation is processed through interfacing with a super-block, Poseidon sub-heaps can be created outside of a process's main thread, which allows a thread to utilize socket-local memory controllers and reduce physical resource contention which is inherent in both the designs of PMDK and Makalu, which require logical sub-heaps to be created in the main thread, thus limiting performance and scalability due to physical resource contention.

Chapter 6

Discussion and Limitations

One limitation is that Poseidon cannot guarantee NVMM heap metadata protection from a malicious 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, an attack 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 [18] and ERIM [42].

We believe that our Poseidon prototype can be further optimized for higher space efficiency and balanced use of hardware resources. For example, our current Poseidon prototype does not distinguish small allocation and large allocation in managing heap metadata, so space overhead for small allocation is relatively higher. We can further reduce space overhead for small allocations by adopting slab allocator design (also known as BIBOP: Big Bag of Pages) storing the same size objects in an allocator page. Other than this, optimal placement of allocator pages among NUMA domains will be an interesting future work of Poseidon, as unbalanced allocation can cause under-utilization of scarce hardware resources, especially NVMM memory controllers which are per-NUMA domain. Thus, maximizing hardware utilization while minimizing remote NVMM access will be an interesting optimization problem.

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 corruption nor prevents persistent memory leak, it can mitigate the side effect of in-place metadata corruption by stopping the propagation of its corruption.

Chapter 7

Related Work

Persistent Memory Allocators. Recently, many studies for persistent allocators have been conducted to achieve scalability and persistency on many-core systems. However, we observe that none of allocators are scalable on a large number of cores and guarantee metadata protection. PMDK allocator libpmemobj [20] 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 serious performance and scalability bottleneck. More importantly, PMDK allocator is exceptionally vulnerable to permanent metadata corruption (persistent memory leaks and/or overallocation of memory regions). Makalu [7] suffers from limited manycore scalability due to several global metadata management structures. Makalu adopts mark-and-sweep garbage collection to discover and fix persistent memory leak without relying micro logging in Poseidon. However, Makalu also fails to deal with metadata corruption. In particular, Makalu is bad because if pointers in an object are corrupted, it will not be able to reclaim all the objects which are reachable by the object. Truly, it is questionable if reachability-based garbage collection is practically useful in memory unsafe languages like C and C++. PAllocator [33] 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. PAllocator does not guarantee metadata safety from program bugs. nvm_malloc [38] is a persistent allocator based on jemalloc [16]. 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 [4] improves performance of applications by maintaining per-processor heaps so as to reduce contention. TCMalloc [17] 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 [16] 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 [27] is another scalable memory allocator focused on achieving low latency with lock-free synchronization.

Secure Memory Allocators. Notably, these state-of-the-art secure allocators [5, 31, 39, 40] mostly adopt a segregated metadata layout. Moreover, heap metadata is stored separately; these allocators 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 safety. This makes their adoption to real-world applications difficult. DieHard [5] 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 randomly allocates pages over the entire possible address space, and carves them up into size-segregated chunks tracked by an allocation bitmap [31]. Both techniques suffers from a high overhead of up to 40% on allocation intensive benchmarks. FreeGuard attempts to combine techniques from both BIBOP (Big Bag of Pages) and freelist allocators [39]. 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 is similar in design to FreeGuard, but is mainly concentrated on randomization-entropy and tunable security guarantees [40]. 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%.

Chapter 8

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 safety. We illustrated how the existing defacto memory allocator (PMDK) is vulnerable to silent data corruption and persistent memory leaks. In order to ensure safety, 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 and real-world, computation-intensive applications. 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.

Bibliography

[1] Anton Altaparmakov. The Linux-NTFS Project, 2005. URL http://linux-ntfs.sourceforge.net/ntfs/,August2005.

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

[3] Arulraj, Pavlo, and Dulloor. Let's Talk About Storage and Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD/PODS Conference*, Melbourne, Victoria, Australia, May 2015.

[4] Berger, Emery, McKinley, Kathryn, Blumofe, Robert, Wilson, and Paul. 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)*, pages 117–128, Cambridge, MA, November 2000.

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

[6] Bhandari, Chakrabarti, and Boehm. 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)*, Amsterdam, Netharlands, October 2016.

[7] Kumud Bhandari, Dhruva R. Chakrabarti, and Hans-J. Boehm. Makalu: Fast re-

coverable allocation of non-volatile memory. In *Proceedings of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 677–694, Amsterdam, Netharlands, October 2016. ACM.

[8] Bhat, Eqbal, Clements, Kaashoek, and Zeldovich. Scaling a file system to many cores using an operation log. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[9] Brown and Avni. PHyTM: Persistent Hybrid Transactional Memory. *PVLDB*, 10: 409–420, 2016.

[10] Mingming Cao, Suparna Bhattacharya, and Ted Ts'o. Ext4: The next generation of ext2/3 filesystem. In *LSF*, 2007.

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

[12] Condit, Nightingale, Frost, Ipek, Lee, Burger, and Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, October 2009.

[13] Intel Corporation. Intel®64 and IA-32 Architectures Software De-veloper's Manual, 2018.

[14] Dong, Bu, Yi, Dong, and Chen. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*, Ontario, Canada, October 2019.

[15] Eckert, Bianchi, Wang, Shoshitaishvili, Kruegel, and Vigna. Heaphopper: bringing bounded model checking to heap implementation security. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

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

[17] Google. Tcmalloc : Thread-caching malloc, 2007. http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

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

[19] Hu, Ren, Badam, Shu, and Moscibroda. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2017.

[20] INTEL. Persistent Memory Development Kit, 2019. URL http://pmem.io/.

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

[22] Izraelevitz, Kelly, and Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[23] 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. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019. URL https://arxiv.org/abs/1903.05714v2.

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

[25] Kolli, Pelley, Saidi, Chen, and Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[26] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 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, February 2015.

[27] Liu, Ran, Chen, and Haibo. 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, July 2012.

[28] Liu, Zhang, Chen, Qian, Wu, and Ren. DUDETM: Building Durable Transactions with Decoupling for Persistent Memory. *ACM SIGPLAN Notices*, 52:329–342, 2017. doi: 10.1145/3093336.3037714.

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

[30] Moraru, Andersen, Kaminsky, Tolia, Ranganathan, and Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. In *TRIOS '13 Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, pages 1–17, 2013. doi: 10.1145/2524211.2524216.

[31] Novark, Gene, Berger, and Emery D. DieHarder: Securing the Heap. In *Proceedings*

of the 17th ACM Conference on Computer and Communications Security (CCS), page 573–584, Chicago, IL, October 2010.

[32] Oukid, Ismael, Booss, Daniel, Lespinasse, Adrien, Lehner, Wolfgang, Willhalm, Thomas, and Gomes. Memory Management Techniques for Large-Scale Persistent-Main-Memory System. *PVLDB*, 10:1166–1177, 2017. doi: 10.14778/3137628.3137629.

[33] Ismail Oukid, Daniel Booss, Adrien Lespinasse, Wolfgang Lehner, Thomas Willhalm, and Grégoire Gomes. 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, August 2017.

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

[35] Phillips and Tan. Exploring Security Vulnerabilities by Exploiting Buffer Overflow using the MIPS ISA. *ACM SIGCSE Bulletin*, 35:172–176, 2003. doi: 10.1145/792548.611962.

[36] Refai. 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, 2006.

[37] Schwalb, Berning, Faust, Dreseler, and Plattner. nvmmalloc: Memory Allocation for NVRAM. In *Proceedings of the International Workshop on Data Management on New Hardware*, Melbourne, Australia, October 2015.

[38] David Schwalb, Tim Berning, Martin Faust†, Markus Dreseler, and Hasso Plattner†.

nvm malloc: Memory Allocation for NVRAM. In *Proceedings of the 41st International Conference on Very Large Data Bases (VLDB)*, Hawaii, USA, September 2015.

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

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

[41] Song and Noh. Towards transparent and seamless storage-as-you-go with persistent memory. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

[42] Vahldiek-Oberwagner, Anjo, Elnikety, Eslam, Duarte, Nuno, Sammler, Michael, Druschel, Peter, Garg, and Deepak. Erim: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Security Symposium (Security)*, pages 1221–1238, Santa Clara, CA, August 2019.

[43] Wang, Jiang, and Xiong. Caching or not: rethinking virtual file system for non-volatile main memory. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.

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

[45] Wright, Charles, Spillane, Richard, Sivathanu, Goplan, Zadok, and Erez. Extending ACID Semantics to the file system. *TOS*, 3, 2007. doi: 10.1145/1242520.1242521.

[46] Xu, Zhang, Memaripour, Gangadharaiah, Borase, Da Silva, Swanson, and Rudoff. NOVA-Fortis: A Fault-Tolerance Non-Volatile Main Memory File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, October 2017.

[47] Jian Xu and Steven Swanson. 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, February 2016.

[48] Yang, Izraelevitz, and Swanson. Orion: a distributed file system for non-volatile main memories and RDMA-capable networks. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST)*, Boston, MA, February 2019.

[49] Zhang and Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, July 2016.

[50] Zhang, Yang, Memaripour, and Swanson. Mojim: A Reliable and Highly-Available Non-Volatile Memory System. *ACM SIGPLAN Notices*, 50:3–18, 2015. doi: 10.1145/2775054.2694370.

[51] Zuo and Hua. SecPM: a secure and persistent memory system for non-volatile memory. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, Oakland, California, USA, February 2018.