Pangolin: A Fault-Tolerant Persistent Memory Programming Library

Lu Zhang and Steven Swanson, *UC San Diego*

https://www.usenix.org/conference/atc19/presentation/zhang-lu

This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.

July 10–12, 2019 • Renton, WA, USA

ISBN 978-1-939133-03-8

Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.

Pangolin: A Fault-Tolerant Persistent Memory Programming Library

Lu Zhang
University of California, San Diego
luzh@eng.ucsd.edu

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

Abstract

Non-volatile main memory (NVMM) allows programmers to build complex, persistent, pointer-based data structures that can offer substantial performance gains over conventional approaches to managing persistent state. This programming model removes the file system from the critical path which improves performance, but it also places these data structures out of reach of file system-based fault tolerance mechanisms (e.g., block-based checksums or erasure coding). Without fault-tolerance, using NVMM to hold critical data will be much less attractive.

This paper presents Pangolin, a fault-tolerant persistent object library designed for NVMM. Pangolin uses a combination of checksums, parity, and micro-buffering to protect an application's objects from both media errors and corruption due to software bugs. It provides these protections for objects of any size and supports automatic, online detection of data corruption and recovery. The required storage overhead is small (1% for gigabyte-sized pools of NVMM). Pangolin provides stronger protection, requires orders of magnitude less storage overhead, and achieves comparable performance relative to the current state-of-the-art fault-tolerant persistent object library.

1 Introduction

Emerging non-volatile memory (NVM) technologies (e.g., battery-backed NVDIMMs [31] and 3D XPoint [30]) provide persistence with performance comparable to DRAM. Non-volatile main memory (NVMM), is byte-addressable, cache-coherent NVM that resides on the system's main memory bus. The combination of NVMM and DRAM enables hybrid memory systems that offer the promise of dramatic increases in storage performance and a more flexible programming model.

A key feature of NVMM is support for direct access, or DAX, that lets applications perform loads and stores directly to a file that resides in NVMM. DAX offers the lowest-possible storage access latency and enables programmers to craft complex, customized data structures for specific applications. To support this model, researchers and industry have proposed various persistent object systems [3, 5, 12, 20, 25, 28, 39, 47].

Building persistent data structures presents a host of challenges, particularly in the area of crash consistency and fault tolerance. Systems that use NVMM must preserve crash-consistency in the presence of volatile caches, out-of-order execution, software bugs, and system failures. To address these challenges, many groups have proposed crash-consistency solutions based on hardware [33, 34, 37, 40], file systems [4, 10, 48, 49], user-space data structures and libraries [3, 5, 12, 39, 43, 47, 50], and languages [9, 38].

Fault tolerance has received less attention but is equally important: To be viable as an enterprise-ready storage medium, persistent data structures must include protection from data corruption. Intel processors report uncorrectable memory media errors via a machine-check exception and the kernel forwards it to user-space as a `SIGBUS` signal. To our knowledge, Xu *et al.* [49] were the first to design an NVMM file system that detects and attempts to recover from these errors. Among programming libraries, only `libpmemobj` provides any support for fault tolerance, but it incurs 100% space overhead, only protects against media errors (not software "scribbles"), and cannot recover corrupted data without taking the object store offline.

Xu *et al.* also highlighted a fundamental conflict between DAX-`mmap()` and file system-based fault tolerance: By design, DAX-`mmap()` leaves the file system unaware of updates made to the file, making it impossible for the file system to update the redundancy data for the file. Their solution is to disable file data protection while the file is mapped and restore it afterward. This provides well-defined protection guarantees but leaves file data unprotected when it is in use.

Moving fault-tolerance to user-space NVMM libraries solves this problem, but presents challenges since it requires integrating fault tolerance into persistent object libraries that manage potentially millions of small, heterogeneous objects.

To satisfy the competing requirements placed on NVMM-based, DAX-mapped object store, a fault-tolerant persistent object library should provide at least the following characteristics:

1. **Crash-consistency**. The library should provide the means to ensure consistency in the face of both system failures and data corruption.
2. **Protection against media and software errors**. Both types of errors are real threats to data stored to NVMM, so the library should provide protection against both.
3. **Low storage overhead**. NVMM is expensive, so minimizing storage overhead of fault tolerance is important.
4. **Online recovery**. For good availability, detection and recovery must proceed without taking the persistent object store offline.
5. **High performance**. Speed is a key benefit of NVMM. If fault-tolerance incurs a large performance penalty, NVMM will be much less attractive.
6. **Support for diverse objects**. A persistent object system must support objects of size ranging from a few cache lines to many megabytes.

This paper describes *Pangolin*, the first persistent object library to satisfy all these criteria. Pangolin uses a combination of parity, replication, and object-level checksums to provide space-efficient, high-performance fault tolerance for complex NVMM data structures. Pangolin also introduces a new technique for accessing NVMM called *micro-buffering* that simplifies transactions and protects NVMM data structures from programming errors.

We evaluate Pangolin using a suite of benchmarks and compare it to `libpmemobj`, a persistent object library that offers a simple replication mode for fault tolerance. Compared to `libpmemobj`, performance is similar, and Pangolin provides stronger protection, online recovery, and greatly reduced storage overhead (1% instead of 100%).

The rest of the paper is organized as follows: Section 2 provides a primer on NVMM programming and NVMM error handling in Linux. Section 3 describes how Pangolin organizes data, manages transactions, and detects and repairs errors. Section 4 presents our evaluations. Section 5 discusses related work. Finally, Section 6 concludes.

2 Background

Pangolin lets programmers build fault-tolerant, crash-consistent data structures in NVMM. This section first introduces NVMM and the DAX mechanism applications use to gain direct access to persistent data. Then, we describe the NVMM error handling mechanisms that Intel processors and Linux provide. Finally, we provide a brief primer on NVMM programming using `libpmemobj` [39], the library on which Pangolin is based.

2.1 Non-volatile Main Memory and DAX

Several technologies are poised to make NVMM common in computer systems. 3D XPoint [30] is the closest to wide deployment. Phase change memory (PCM), resistive RAM (ReRAM), and spin-torque transfer RAM (STT-RAM) are also under active development by memory manufacturers. Flash-backed DRAM is already available and in wide use. Linux and Windows both have support for accessing NVMM and using it as storage media.

The performance and cost parameters of NVMM lie between DRAM and SSD. Its write latency is longer than DRAM, but it will cost less per bit. From the storage perspective, NVMM is faster but more expensive than SSD.

The most efficient way to access NVMM is via direct access (DAX) [15] memory mapping (i.e., DAX-`mmap()`). To use DAX-`mmap()`, applications map pages of a file in an NVMM-aware file system into their address space, so the application can access persistent data from the user-space using load and store instructions, without the file system intervening.

2.2 Handling NVMM Media Errors

To recover from data corruption, Pangolin relies on error detection and media management facilities that the processor and operating system provide together. Below, we describe these facilities available on Intel and Linux platforms. Windows provides similar mechanisms.

Hardware Error Correction Memory controllers for commercially available NVMMs (i.e., battery-backed DRAM and 3D XPoint) implement error-correction code (ECC) in hardware to detect and correct media errors when they can, and they report uncorrectable (but detectable) errors with a machine check exception (MCE) [14] that the operating system can catch and attempt to handle.

Pangolin provides a layer of protection in addition to the ECC hardware provides, but it does not require hardware ECC. Pangolin uses checksums to detect errors that hardware cannot detect. This mechanism also catches software bugs (which are invisible to hardware ECC). ECC does, however, improve performance by transparently handling many media errors.

Regardless of the ECC algorithm hardware provides, field studies of DRAM and SSDs [13, 29, 35, 41, 42, 45] have shown that detectable but uncorrectable media errors occur frequently enough to warrant additional software protection. Furthermore, file systems [23, 49, 51] apply checksums to

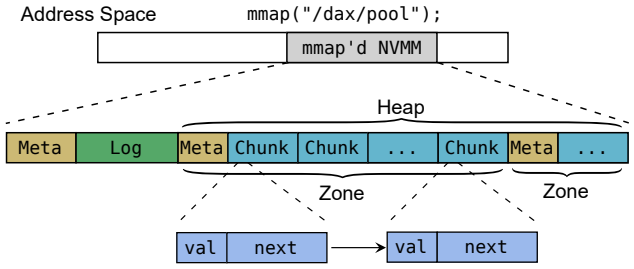

Figure 1: DAX-mapped NVMM as an object store – `Libpmemobj` divides the mapped space into zones and chunks for memory management. The `val` field is a 64-bit integer and the `next` field is a persistent pointer (`PMEMoid`) pointing to the next node object in the pool.

```
1  PMEMobjpool *pool = pmemobj_open("/dax/pool");
2  ...
3  struct node *n = pmemobj_direct(node_oid);
4  n->val = value;
5  pmemobj_persist(pool, &n->val, 8);
6  ...
7  TX_BEGIN(pool) {
8    n = pmemobj_direct(node_oid);
9    pmemobj_tx_add_range(node_oid, 0, sizeof(*n));
10   n->next = pmemobj_tx_alloc(...);
11 } TX_ONABORT {
12   /* handling transaction aborts */
13 } TX_END
14 ...
15 pmemobj_close(pool);
```

Listing 1: A `libpmemobj` program - First modify a node value in a linked list, and later allocate and link a new node from the pool.

their data structures to protect against scribbles.

Repairing Errors When the hardware detects an uncorrectable error, the Linux kernel marks the region surrounding the failed load as "poisoned," and future loads from the region will fail with a bus error. Pangolin assumes an error poisons a 4 KB page since Linux currently manages memory failures at page granularity.

If a running application causes an MCE (by loading from a poisoned page), the kernel sends it a `SIGBUS` and the application can extract the affected address from the data structure describing the signal.

The software can repair the poisoned page by writing new data to the region. In response, the operating system and NVDIMM firmware work together to remap the poisoned addresses to functioning memory cells. The details of this process are part of the Advanced Configuration and Power Interface (ACPI) [46] for NVDIMMs.

Recent kernel patches [6–8, 26] and NVMM library [39] provide utilities for user-space applications to restore lost data by re-writing affected pages with recovered contents (if available).

2.3 NVMM Programming

In this section, we describe `libpmemobj`'s programming model. `Libpmemobj` is a well-supported, open-source C library for programming with DAX-mapped NVMM. It provides facilities for memory management and software transactions that let applications build a persistent object store. Pangolin's interface and implementation are based on `libpmemobj` from PMDK v1.5.

Linux exposes NVMM to the user-space as memory-mapped files (Figure 1). `Libpmemobj` (and Pangolin) refer to the mapped file as a *pool* of persistent objects. Each pool spans a continuous range of virtual addresses.

Within a pool, `libpmemobj` reserves a metadata region that contains information such as the pool's identification (64-bit UUID) and the offset to a "root object" from which all other live objects are reachable. Next, is an area reserved for transaction logs. `Libpmemobj` uses redo logging for its metadata updates and undo logging for application object updates. Transaction logs reside in one of two locations depending on their sizes. Small log entries live in the provisioned "Log" region, as shown in Figure 1. Large ones overflow into the "Heap" storage area.

The rest of the pool is the persistent heap. `Libpmemobj`'s NVMM allocator (a persistent variant of `malloc`/`free`) manages it. The allocator divides the heap's space into several "zones" as shown in Figure 1. A zone contains metadata and a sequence of "chunks." The allocator divides up a chunk for small objects and coalesces adjacent chunks for large objects. By default, a zone is 16 GB, and a chunk is 256 KB.

Listing 1 presents an example to highlight the key concepts of NVMM programming. The code performs two independent operations on a persistent linked list: one is to modify a node's value, and another is to allocate and link a new node.

This example demonstrates two styles of crash-consistent NVMM programming: *atomic*-style (lines 3-5) for a simple modification that is 8 bytes or smaller, and *transactional*-style (lines 7-13) for arbitrary-sized NVMM updates.

Building data structures in NVMM using `libpmemobj` (or any other persistent object library) differs from conventional DRAM programming in several ways:

Memory Allocation `Libpmemobj` provides crash-consistent NVMM allocation and deallocation functions: `pmemobj_tx_alloc`/`pmemobj_tx_free`. They let the programmer specify object type and size to allocate and prevent orphaned regions in the case of poorly-time crashes.

Addressing Scheme Persistent pointers within a pool must remain valid regardless of at what virtual address the pool resides. `Libpmemobj` uses a `PMEMoid` data structure to address an object within a pool. It consists of a 64-bit file ID

and a 64-bit byte offset relative to the start of the file. The `pmemobj_direct()` function translates a `PMEMoid` into a native pointer for use in load or store instructions.

Failure-atomic Updates Modern x86 CPUs only guarantee that 8-byte, aligned stores atomically update NVMM [16]. If applications need larger atomic updates, they must manually construct software transactions. `Libpmemobj` provides undo log-based transactions. The application executes stores to NVMM between the `TX_BEGIN` and `TX_END` macros, and snapshots (`pmemobj_tx_add_range`) a range of object data before modifying it in-place.

Persistence Ordering Intel CPUs provide cache flush/write-back (e.g., `CLFLUSH(OPT)` and `CLWB`) and memory ordering (e.g., `SFENCE`) instructions to make guarantees about when stores become persistent. In Listing 1, the `pmemobj_persist` function and `TX` macros integrate these instructions to flush modified object ranges.

`Libpmemobj` supports a replicated mode that requires a replica pool, doubling the storage the object store requires. `Libpmemobj` applies updates to both pools to keep them synchronized.

Replicated `libpmemobj` can detect and recover from media errors only when the object store is offline, and it cannot detect or recover from data corruption caused by errant stores to NVMM – so-called "scribbles," that might result from a buffer overrun or dereferencing a wild pointer.

3 Pangolin Design

Pangolin allows programmers to build complex, crash-consistent persistent data structures that are also robust in the face of media errors and software "scribbles" that corrupt data. Pangolin satisfies all of the criteria listed in Section 1. This section describes its architecture and highlights the key challenges that Pangolin addresses to meet those requirements. In particular, Pangolin provides the following features unseen in prior works.

- It provides fast, space-efficient recovery from media errors and scribbles.
- It uses checksums to protect object integrity and supports incremental checksum updates.
- It integrates parity and checksum updates into an NVMM transaction system.
- It periodically scrubs data to identify corruption.
- It detects and recovers from media errors and scribbles online.

Pangolin guarantees that it can recover from the loss of any single 4 KB page of data in a pool. In many cases, it can recover from the concurrent loss of multiple pages.

We begin by describing how Pangolin organizes data to protect user objects, library metadata, and transaction logs using a combination of parity, replication, and checksums. Next, we describe micro-buffers and explain how they allow Pangolin to preserve a simple programming interface and protect against software scribbles. Then, we explain how Pangolin detects and prevents NVMM corruption and elaborate on Pangolin's transaction implementation with support for efficient, concurrent updates of object parity. Finally, we discuss how Pangolin restores data integrity after corruption and crashes.

3.1 Pangolin's Data Organization

Pangolin uses replication for its internal metadata and RAID-style parity for user objects to provide redundancy for corruption recovery. The MCE mechanism described in Section 2.2 and object checksums in Pangolin detect corruption.

Pangolin views a zone's chunks as a two-dimensional array as shown in the middle of Figure 2. Each *chunk row* contains multiple, contiguous chunks and the chunks "wrap around" so that the last chunk of a row and the first chunk of the next are adjacent. Pangolin reserves the last chunk row for parity.

In our description of Pangolin, we define a *page column* as a one page-wide, aligned column that cuts across the rows of a zone. A *range column* is similar, but can be arbitrarily wide (no more than a chunk row's size).

Initializing a parity-coded NVMM pool requires zeroing out all the bytes in the file. This is a one-time overhead when creating a pool file and does not affect run-time performance. We report this latency in Section 4.

To detect corruption in user objects, Pangolin adds a 32-bit checksum to the object's header. The header also contains the object's size (64-bit) and type (32-bit). The compiler determines type values according to user-defined object types. Pangolin inherits this design from `libpmemobj` and changes the type identifier from 64-bit to 32-bit for the checksum.

Pangolin's object placement is independent of chunk and row boundaries. Objects can be anywhere within a zone, and they can be of any size (up to the zone size).

In addition to user objects, the library maintains metadata for the pool, zones, and chunks, including allocation bitmaps. Pangolin checksums these data structures to detect corruption and replicates the pool's and zones' metadata for fault tolerance. These structures are small (less than 0.1% for pools larger than 1 GB), so replicating them is not expensive. Pangolin uses zone parity to support recovery of chunk metadata.

Pangolin checksums transaction logs and replicates them for redundancy. It treats log entries in zone storage as zeros during parity calculations. This prevents parity update contention between log entries and user objects (see Section 3.5).

Fault Tolerance Guarantees Pangolin can tolerate a single 4 KB media error anywhere in the pool, regardless of whether it is a data page or a parity page. Based on the bad

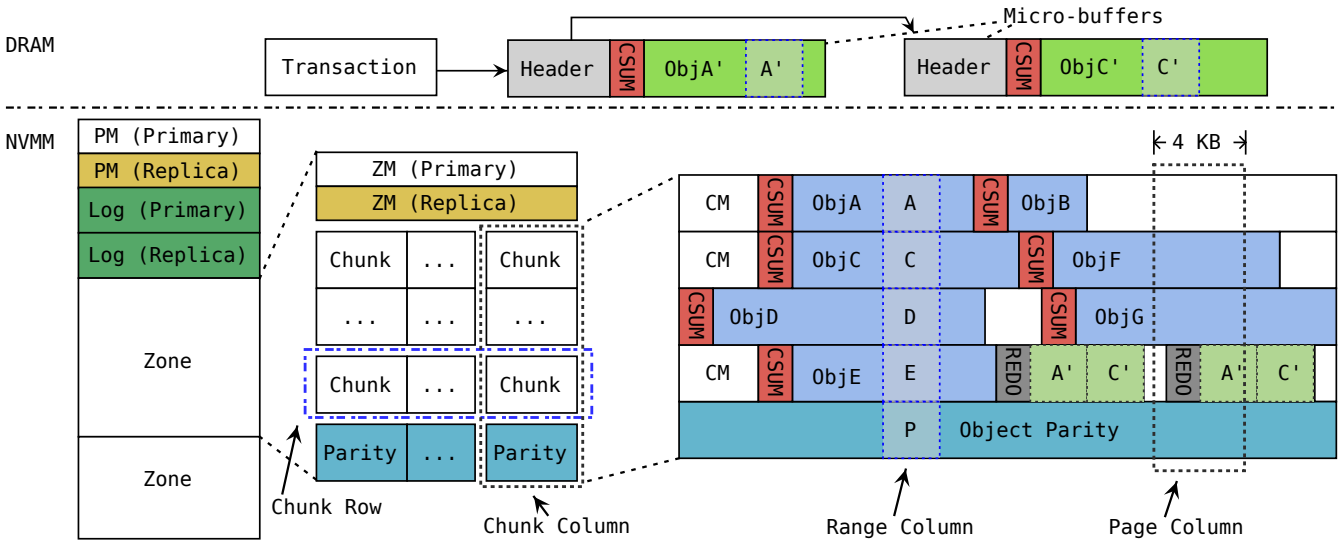

Figure 2: Data protection scheme in Pangolin – Pangolin protects pool metadata (PM), zone metadata (ZM), and chunk metadata (CM). In the highlighted range column, $P = A \oplus C \oplus D \oplus E$. One thread's transaction is modifying ranges A and C of two objects. Pangolin keeps modified data in redo log entries (checksummed and replicated) when the transaction commits. The DRAM part shows micro-buffers for the two objects.

page's address Pangolin can locate its page column and restore its data using other healthy pages.

Faults affecting two pages of the same page column may cause data loss if the corrupted ranges overlap. If an application demands more robust fault tolerance, it can increase the chunk row size, reducing the number of rows and, consequently, the likelihood that two corrupt pages overlap.

Pangolin can recover from scribbles (contiguous overwrites caused by software errors) on NVMM data up to a chunk-row size. By default, Pangolin uses 100 chunk rows, and parity consumes ~1% of a pool's size (e.g., 1 GB for a 100 GB pool).

3.2 Micro-buffering for NVMM Objects

Pangolin introduces micro-buffering to hide the complexity of updating checksums and parity when modifying NVMM objects. Adding checksums to objects and protecting them with parity makes updates more complex, since all three – object data, checksum, and parity – must change at once to preserve consistency. This challenge is especially acute for the atomic programming model as shown in Listing 1 (line 3-5) because a single 8-byte NVMM write cannot host all these updates.

Micro-buffering creates a shadow copy of an NVMM object in DRAM, which separates an object's transient and persistent versions (Figure 2). In Listing 2, `pgl_open` creates a micro-buffer for the node object by allocating a DRAM buffer and copying the node's data from NVMM. It also verifies the object's checksum and performs corruption recovery if necessary.

```
1 struct node *n = pgl_open(node_oid);
2 n->val = value;
3 pgl_commit(pool, n);
```

Listing 2: A Pangolin transaction for a single-object - This snippet corresponds to line 3-5 of Listing 1.

The application can modify the micro-buffered object without concern for its checksum, parity, and crash-consistency because changes exist only in the micro-buffer. When the updates finish, `pgl_commit` starts a transaction that atomically updates the NVMM object, its checksum, and parity (described below). Compared to line 3-5 of Listing 1, Pangolin retains the simple, atomic-style programming model for modifying a single NVMM object, and it supports updates within an object beyond 8 bytes.

Each micro-buffer's header contains information such as its NVMM address, modified ranges, and status flags (e.g., allocated or modified). We elaborate on Pangolin's programming interface and how to construct complex transactions with micro-buffering in Section 3.4.

Another important consideration for micro-buffering is to prevent misbehaving software from corrupting NVMM. If an application's code can directly write to NVMM, as `libpmemobj` allows to, software bugs such as buffer overflows and using dangling pointers can easily cause NVMM

Function	Semantics
`pgl_tx_begin()/commit()/end(), etc.`	Control the lifetime of a Pangolin transaction.
`pgl_tx_alloc()/free()`	Allocate or deallocate an NVMM object.
`pgl_tx_open(PMEMoid oid, ...)`	Create a thread-local micro-buffer for an NVMM object. Verify (and restore) the object integrity, and return a pointer to the micro-buffered user object.
`pgl_tx_add_range(PMEMoid oid, ...)`	Invoke `pgl_tx_open` and then mark a range of the object that will be modified.
`pgl_get(PMEMoid oid)`	Get access to an object, either directly in NVMM or in its micro-buffer, depending on the transaction context. By default, it does not verify the checksum.
`pgl_open(PMEMoid oid, ...)`	Create a micro-buffer for an NVMM object without a transaction. Check the object integrity, and return a pointer to the micro-buffered user object.
`pgl_commit(void *uobj)`	Automatically start a transaction and commit the modified user object in a micro-buffer to NVMM.

Table 1: The Pangolin API - Pangolin's interface mirrors `libpmemobj`'s except that Pangolin does not allow direct writing to NVMM. Pangolin provides single-object transactions using `pgl_open` and `pgl_commit` to convert application code using `libpmemobj`'s atomic updates.

corruption. Conventional debugging tools for memory safety, such as Valgrind [36] and AddressSanitizer [44], insert inaccessible bytes between objects as "redzones" to trap illegal accesses. This approach fails to work for directly accessed NVMM objects because once they are allocated, there is no guarantee for spacing between them, and thus, redzones may land on a nearby, accessible object. One viable approach to using these tools is to let the NVMM allocator insert redzones. However, the presence of redzone bytes will pollute the pool and may exacerbate fragmentation.

Using micro-buffers isolates transient writes from persistent data, and since micro-buffers are dynamically allocated using `malloc()`, they are compatible with existing memory debugging tools. Without using debugging tools, Pangolin also protects micro-buffers by inserting a 64-bit "canary" word in each micro-buffer's header and checks its integrity before writing back to NVMM. On transaction commit, if Pangolin detects a canary mismatch, it aborts the transaction to avoid propagating the corruption to NVMM. Pangolin uses checksums to detect corruptions that may bypass the canary protection.

3.3 Detecting NVMM Corruption

Pangolin uses three mechanisms to detect NVMM corruption. First, it installs a handler for `SIGBUS` (see Section 2.2) that fires when the Linux kernel receives an MCE. A signal handler has access to the address the offending load accessed, and Pangolin can determine what kind of data (i.e., metadata or a user object) lives there and recover appropriately. This mechanism detects media failures, but it cannot discover corrupted data caused by software "scribbles."

To detect scribbles, Pangolin verifies the integrity of user objects using their checksums. Verifying checksums on every access can be expensive. To limit this cost, by default Pangolin only verifies checksums during micro-buffer creation before any object is modified in a transaction. This keeps Pangolin from recalculating a new checksum based on corrupt data. For read-only objects that are accessed by `pgl_get` without micro-buffering, by default Pangolin does not verify checksums. To protect them, Pangolin provides two alternative operation modes: "Scrub" mode runs a scrubbing thread that verifies and restores the whole pool's data integrity when a preset number of transactions have completed, and "Conservative" mode verifies the checksum for every object access (including `pgl_get`). We evaluate the impact of different checksum verification policies in Section 4.

Finally, Linux keeps track of known bad pages of NVMM across reboots. When opening a pool or during its scrubbing, Pangolin can extract this information and recover the data in the reported pages (not currently implemented).

3.4 Fault-Tolerant Transactions

Failure-atomic transactions are central to Pangolin's interface, and they must include verification of data integrity and updates to the checksums and parity data that protect objects. Table 1 summarizes Pangolin's core functions.

Pangolin supports arbitrary-sized transactions and we have made similar APIs and macros as `libpmemobj`'s. The program in Listing 1 can be easily transformed to Pangolin using equivalent functions. One subtle difference is in the handling of atomic-style updates, as shown in Listing 2.

In Pangolin, each thread can execute one transaction or nested transactions (same as `libpmemobj`). Concurrent transactions can execute if each one is associated with a different thread. Currently, Pangolin does not allow concurrent transactions to modify the same NVMM object. Concurrently modifying a shared object may cause data inconsistency if one transaction has to abort. `Libpmemobj` has the same limitation [17].

Each transaction manages its own micro-buffers using a

thread-local hashmap [24], indexed by an NVMM object's `PMEMoid`. Therefore, in a transaction, calling `pgl_tx_open` for the same object either creates or retrieves its micro-buffer. Multiple micro-buffers opened in one transaction form a linked list as shown in Figure 2. Micro-buffers for one transaction are not visible in other transactions, providing isolation.

If a transaction modifies an object, Pangolin copies it to a micro-buffer, performs the changes there, and then propagates the changes to NVMM during commit. Since changes occur in DRAM (which does not require undo information), Pangolin implements redo logging.

At transaction commit, Pangolin recomputes the checksums for modified micro-buffers, creates and replicates redo log entries for the modified parts of the micro-buffers and writes these ranges back to NVMM objects. Then, it updates the affected parity bits (see Section 3.5) and marks the transaction committed. Finally, Pangolin garbage-collects its logs and closes thread-local micro-buffers.

If a transaction aborts, either due to unrecoverable data corruption or other run-time errors, Pangolin discards the transaction's micro-buffers without touching NVMM.

A transaction can also allocate and deallocate objects. Pangolin uses redo logging to record NVMM allocation and free operations, just as `libpmemobj` does.

For read-only workloads, repeatedly creating micro-buffers and verifying object checksums can be very expensive. Therefore, Pangolin provides `pgl_get` to gain direct access to an NVMM object without verifying the object's checksum. The application can verify an object's integrity manually as needed or rely on Pangolin's periodic scrubbing mechanism. Inside a transaction context, `pgl_get` returns a pointer to the object's micro-buffer to preserve isolation.

3.5 Parity and Checksum Updates

Objects in different rows can share the same range of parity, and we say these objects *overlap*. Object overlap leads to a challenge for updating the shared parity because updates from different transactions must serialize but naively locking the whole parity region sacrifices scalability.

For instance, using *ObjA* and *ObjC* in Figure 2, suppose two different transactions modify them, replacing A with A' and C with C', respectively. After both transactions update P, the parity should have the value $P' = A' \oplus C' \oplus D \oplus E$ regardless of how the two transaction commits interleave.

Pangolin uses a combination of two techniques that exploit the commutativity of XOR and fine-grained locking to preserve correctness and scalability.

Atomic parity updates The first approach uses the atomic XOR instruction (analogous to an atomic increment) that modern CPUs provide to perform incremental parity updates for changes to each overlapping object.

In our example, we can compute two parity patches: $\Delta A = A \oplus A'$, $\Delta C = C \oplus C'$ and then rewrite P' as $P \oplus \Delta A \oplus \Delta C$. Since XOR commutes and is a bit-wise operation, the two threads can perform their updates without synchronization.

Hybrid parity updates Atomic XOR is slower than normal or vectorized XOR. For small updates, the latency difference between them is not significant, and Pangolin prefers atomic XOR instructions to update parity without the need for locks. But for large parity updates, atomic XOR can be inefficient. Therefore, Pangolin's hybrid parity scheme switches to vectorized XOR for large transfers.

To coordinate large and small parity updates, Pangolin uses *parity range-locks*, that work similarly as reader/writer locks (or shared mutex): Small writes take shared ownership of a range lock and update parity with atomic XOR instructions. Large updates using vectorized XORs take exclusive ownership of a range-lock, and only one thread can modify parity in a locked range. If one update involves multiple range-locks, serialization happens on a per-range-lock basis.

The managed size of a parity range-lock depends on the performance trade-off between Pangolin's parity mode and `libpmemobj`'s replication mode. We discuss this in Section 4.

Pangolin refreshes an object's checksum in its micro-buffer before updating parity, and it considers the checksum field as one of the modified ranges of the object. Checksums like CRC32 requires recomputing the checksum using the whole object. This can become costly with large objects. Thus, Pangolin uses Adler32 [23], a checksum that allows incremental updates, to make the cost of updating an object's checksum proportional to the size of the modified range rather than the object size.

We implement Pangolin's parity and checksum updates using the Intelligent Storage Acceleration Library (ISA-L) [18], which leverages SIMD instructions of modern CPUs for these data-intensive tasks.

Protections for other transaction systems Other NVMM persistent object systems could apply Pangolin's techniques for parity and checksum updates. For example, consider an undo logging (as opposed to Pangolin's redo logging) system that first stores a "snapshot" copy of an object in the log before modifying the original in-place. In this case, the system could compute a parity patch using the XOR result between the logged data (old) and the object's data (new). Then, it can apply the parity patch using the hybrid method we described in this section.

3.6 Recovering from Faults

In this section, we discuss how Pangolin recovers data integrity from both NVMM corruption and system crashes.

Corruption recovery Pangolin uses the same algorithm to recover from errors regardless of how it detects them (i.e., via `SIGBUS` or a checksum mismatch).

The first step is to pause the current thread's transaction, and to wait until all other outstanding transactions have com-

pleted. Meanwhile, Pangolin prevents the initialization of new transactions by setting the pool's "freeze" flag. This is necessary because, during transaction committing, parity data may be inconsistent.

Once the pool is frozen, Pangolin uses the parity bits and the corresponding parts of each row in the page column to recover the missing data.

Pangolin preserves crash-consistency during repair by making persistent records of the bad pages under recovery. Recovery is idempotent, so it can simply re-execute after a crash.

Pangolin's current implementation only allows one thread to perform any online corruption recovery, and if the thread is executing a transaction, online recovery only works if the thread has not started committing. If two threads encounter faults simultaneously, Pangolin kills the application and performs post-crash recovery (see below) when it restarts. Supporting multi-threaded online recovery, and allowing it to work when threads have partially written NVMM is possible, but it requires complex reasoning about how to restore the data and its parity to a consistent state.

Crash recovery Pangolin handles recovery from a crash using its redo logs. It must also protect against the possibility that the crash occurred during a parity update.

To commit a transaction, Pangolin first ensures its redo logs are persistent and replicated, and then updates the NVMM objects and their parity. If a crash happens before redo logs are complete, Pangolin discards the redo logs on reboot without touching the objects or parity. If redo logs exist, Pangolin replays them to update the objects and then recomputes any affected parity ranges using the data written during replay (which is now known to be correct) and the data from the other rows.

Pangolin does not log parity updates because it would double the cost of logging. This does raise the possibility of data loss if a crash occurs during a parity update and a media error then corrupts data of the same page column before recovery can complete. This scenario requires the simultaneous loss of two pages in the same page column due to corruption and a crash, which we expect to be rare.

4 Evaluation

In this section, we evaluate Pangolin's performance and the overheads it incurs by comparing it to normal `libpmemobj` and its replicated version. We start with our experimental setup and then consider its storage requirements, latency impact, scalability, application-level performance, and corruption recovery.

4.1 Evaluation Setup

We perform our evaluation on a dual-socket platform with Intel's Optane DC Persistent Memory [19]. The CPUs are 24-core engineering samples of the Cascade Lake generation. Each socket has 192 GB DDR4 DRAM and 1.5 TB NVMM. We configure the persistent memory modules in *AppDirect* mode and run experiments on one socket using its local DRAM and NVMM. A recent report [21] studying this platform provides more architectural details.

Pmemobj	`libpmemobj` baseline from PMDK v1.5
Pangolin	Pangolin baseline w/ micro-buffering only
Pangolin-ML	Pangolin + metadata and redo log replication
Pangolin-MLP	Pangolin-ML + object parity
Pangolin-MLPC	Pangolin-MLP + object checksums
Pmemobj-R	`libpmemobj` w/ one replication in another file

Table 2: Library operation modes for evaluation - In the figures, we abbreviate Pangolin as `pgl`.

The CPU provides the `CLWB` instruction for writing-back cache lines to NVMM, non-temporal store instructions to bypass caches, and the `SFENCE` instruction to ensure persistence and memory ordering. It also has atomic XOR and AVX instructions that our parity and checksum computations use.

The evaluation machine runs Fedora 27 with a Linux kernel version 4.13 built from source with the NOVA [48] file system. We run experiments with both Ext4-DAX [27] and NOVA, and applications use `mmap()` to access NVMM-resident files. The performance is similar on the two file systems because DAX-`mmap()` essentially bypasses them.

On our evaluation machine, we found that updating parity with atomic XORs becomes worse than `libpmemobj`'s replication mode when the modified parity range is greater than 8 KB, so we set 8 KB as the threshold to switch between those parity calculation strategies (see Section 3.5).

Table 2 describes the operation modes for our evaluations. The Pangolin baseline implements transactions with micro-buffering. It uses buffer canaries to prevent corruption from affecting NVMM, but it does not have parity or checksum for NVMM data.

We evaluate versions of Pangolin that incrementally add metadata and log replication ("+ML"), object parity ("+MLP"), and checksums ("+MLPC"). We combine the impact of metadata updates with log replication because metadata updates are small and cheap in our evaluation.

Pmemobj-R is the replication mode of `libpmemobj` that mirrors updates to a replica pool during transaction commit. Comparing Pangolin-MLP and Pmemobj-R is especially useful because the two configurations protect against the same types of data corruption: media errors but not scribbles.

4.2 Memory Requirements

We discuss and evaluate Pangolin's memory requirements for both NVMM and DRAM.

NVMM All our Pangolin experiments use a single pool of 100 GB that contains 6× 16 GB zones. Pangolin replicates

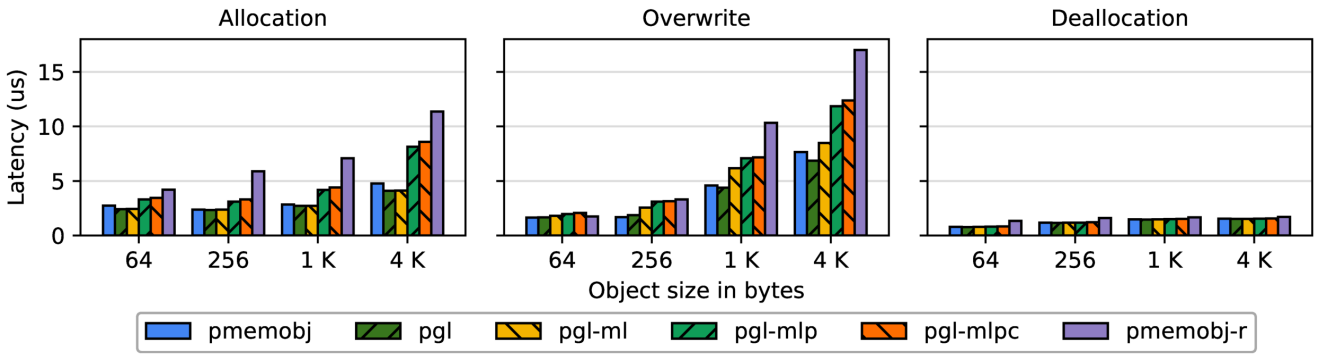

Figure 3: Transaction performance – Each transaction allocates, overwrites, or frees one object of varying sizes. Pangolin's latencies are similar to Pmemobj's. Pangolin-MLP is mostly better than Pmemobj-R because updating parity using atomic XOR and `CLWB` instructions is faster than mirroring data in a separate file.

all the pool's metadata in the same file, which occupies a fixed ∼20 MB. The rest of the space is for user objects and their protection data. By default, Pangolin uses 100 chunk rows, so each zone has about 160 MB parity, and that totally occupies ∼1% of the pool's capacity. Pmemobj-R uses a second 100 GB file as the replica, doubling the cost of NVMM space requirement.

When using parity, Pangolin has to zero out the whole pool to ensure all zones are initially parity-coded. This takes about 130 seconds. It is a one-time overhead for creating the pool and excluded from the following evaluations.

DRAM Pangolin uses `malloc()`'d DRAM to construct micro-buffers. The required DRAM space is proportional to ongoing transaction sizes. Table 3 summarized the transaction sizes for the evaluated key-value store data structures. Pangolin automatically recycles them on transaction commits. In our evaluation experiments, micro-buffering never exceeds using 50 MB of DRAM.

4.3 Transaction Performance

Figure 3 illustrates the transaction latencies for three basic operations on an NVMM object store: object allocation, overwrite, and deallocation. Each transaction operates on one object, and we vary the size of the object.

For allocation, latency grows with object size for all five configurations, due to constructing the object and cache line write-back latency. Pangolin incurs 2% - 13% lower latencies than Pmemobj due to its use of non-temporal stores for write backs. An allocation operation does not involve object logging, so Pangolin-ML shows performance similar to Pangolin. Pangolin-MLP adds overhead to update the parity data. It outperforms Pmemobj-R by between 1.2× and 1.9×. We found this is because updating parity using atomic XORs and `CLWB`s incurs less latency than mirroring data in a separate file, as Pmemobj-R does.

Adding checksum (Pangolin-MLPC) incurs less than 7% overhead compared to Pangolin-MLP. Parity's impact is larger than checksum's because updating a parity range demands values from three parts: the micro-buffer, the NVMM object, and the old parity data, while computing a checksum only needs data in a DRAM-based micro-buffer. Moreover, Pangolin needs to flush the modified parity range to persistence, which is the same size as the object. In contrary, updating a checksum only writes back a single cache line that contains the checksum value.

Overwriting an NVMM object involves transaction logging for crash consistency. Pangolin and Pmemobj store the same amount of logging data in NVMM, although they use redo logging and undo logging for this purpose, respectively. Since log entry size is proportional to an object's modified size, which is the whole object in this evaluation, this cost grows with the object. With Pangolin, log replication accounts for between 7% to 25% of the latency. Parity updates consume between 8% to 27% of the extra latency, depending on object size, and checksum updates account for less than 5%. Pangolin-MLP's performance for overwrites is 12% worse than Pmemobj-R for 64 B object updates and is between 1.1× and 1.5× better than Pmemobj-R for objects larger than 64 B.

Deallocation transactions only modify metadata, so their latencies do not change much.

4.4 Scalability

Figure 4 measures Pangolin's scalability by randomly overwriting existing NVMM objects and varying the object sizes and the number of concurrent threads.

Pangolin uses reader/writer locks to implement the hybrid parity update scheme described in Section 3.5. The number of rows in a zone and the zone size determine the granularity

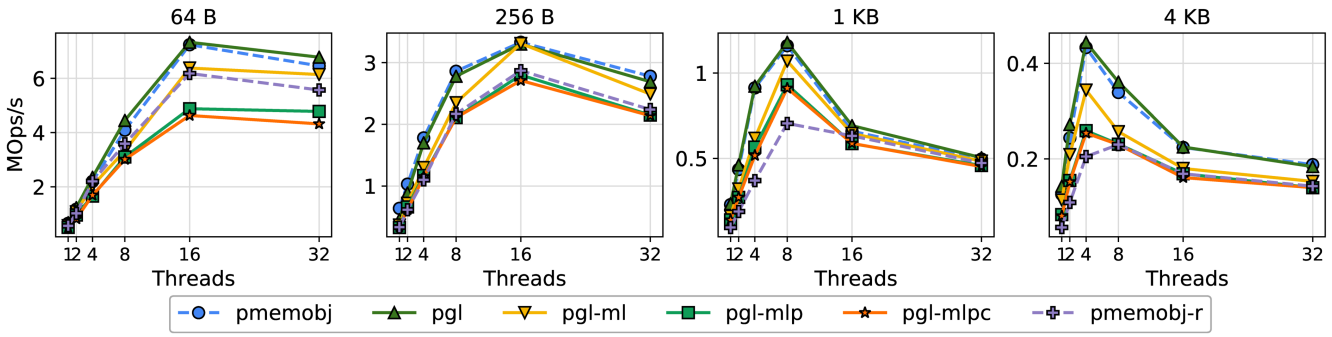

Figure 4: Scalability – Concurrent workloads randomly overwrite objects of varying sizes. Pangolin-MLP scales as well as Pmemobj-R or better for objects larger than 64 B. For 64 B objects, Pangolin-MLP is worse than Pmemobj-R by between 6% and 25% due to synchronization overhead for online recovery.

		ctree	rbtree	btree	skiplist	rtree	hashmap
Object size		56	80	304	408	4136	10 M (table), 40 (entry)
Insert	New	56 (1.00)	80 (1.00)	65.9 (0.22)	408 (1.00)	4502 (1.09)	60.9 (1.00)
	Mod	127.6 (3.28)	330.2 (5.13)	381.2 (1.47)	33.9 (2.50)	200.0 (5.05)	331.1 (4.21)
Remove	New	0	0	0	0	184.1 (0.05)	10.5 (1×10^{-5})
	Mod	28.0 (0.50)	202.8 (2.65)	268.3 (0.90)	16.9 (0.75)	98.6 (2.52)	254.3 (2.16)

Table 3: Data structure and transaction sizes - "Insert" and "Remove" show average transaction sizes for insertions and removals, respectively. "New" and "Mod" indicate average allocated and modified sizes. Value in parentheses is the average number of objects involved in the transaction.

of these locks: For a fixed zone size, more rows means fewer columns and fewer parity range-locks.

There is no lock contention in the results because the transactions use atomic XOR instructions and can execute concurrently (only taking the reader locks). Our configuration with 1% parity (160 MB parity per 16 GB zone) has 20 K range-locks per zone, so the chance of lock contention is slim even with large updates (more than 8 KB) and many cores.

The graphs also show how each Pangolin's fault-tolerance mechanisms affect performance. Pangolin's throughput is very close to Pmemobj. Pangolin-MLP mostly outperforms Pmemobj-R for object updates that are 256 B or larger, up to 1.5×. But for 64 B object updates, it performs worse than Pmemobj-R by between 6% and 25%. This is because when enabling parity, every Pangolin transaction checks the pool freeze flag (an atomic variable), incurring synchronization overhead. This overhead is noticeable for short transactions with 64 B objects but becomes negligible for larger updates. Pangolin-MLPC only performs marginally worse than Pangolin-MLP.

Scaling degrades for all configuration as update size and thread count grow because the sustainable bandwidth of the persistent memory modules becomes saturated.

4.5 Impacts on NVMM Applications

To evaluate Pangolin in more complex applications, we use six data structures included in the PMDK toolkit: crit-bit tree (ctree), red-black tree (rbtree), btree, skiplist, radix tree (rtree), and hashmap. They have a wide range of object sizes and use a diverse set of algorithms to insert, remove, and lookup values. We rewrite these benchmarks with Pangolin's programming interface as described in Section 3.4.

Table 3 summarizes the object and transaction sizes for each workload. The tree structures and the skiplist have a single type of object, which is the tree or list node. Hashmap has two kinds of objects. One is the hash table that contains pointers to buckets. The hash table grows as the application inserts more key-value pairs. Each bucket is a linked list of fixed-sized entry objects.

Each insertion or removal is a transaction processing a key-value pair. The workloads involve a mix of object allocations, overwrites, and deallocations. Table 3 shows, on average, the number of bytes and objects (in parentheses) involved in each data structure's transaction. Deallocated sizes are not shown because they marginally affect the performance differences (see Figure 3).

An average allocation size ("New" rows in the table) smaller than the object size means the data structure does not allocate a new object for every insert operation (e.g., btree).

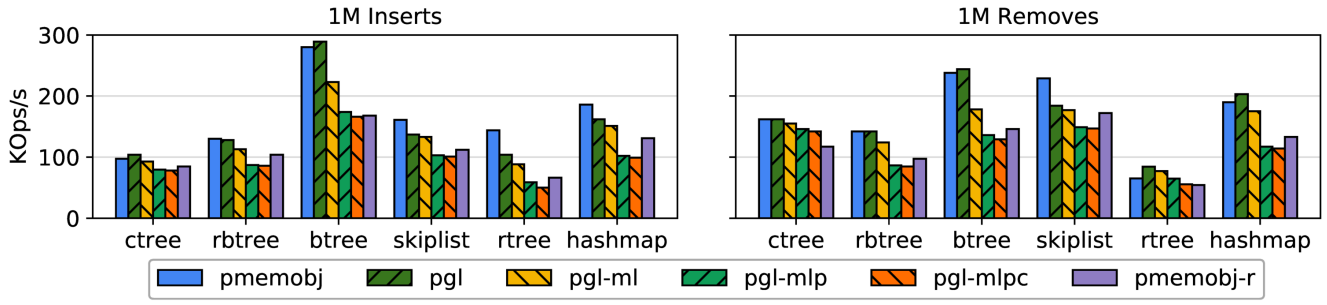

Figure 5: Key-value store performance – Each transaction either inserts or removes one key-value pair from the data store. Pangolin performs similarly to Pmemobj except for cases when a transaction's modified size is much less than an object's size (e.g., skiplist and rtree) due to micro-buffering overhead. Pangolin-MLP's performance is 95% of Pmemobj-R on average.

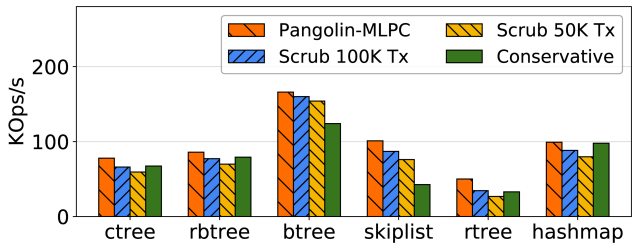

Figure 6: Checksum verification impact – Pangolin-MLPC bars are the same as those in Figure 5 for 1M Inserts. The cost of different policies depends strongly on data structures.

	ctree	rbtree	btree	sklist	rtree	hmap
Pmemobj	1.0	1.0	1.0	1.0	1.0	1.0
Pgl-MLPC	0.92	0.84	0.87	0.96	0.42	0.42
Scrub 100K	0.10	0.09	0.09	0.10	0.04	0.05
Scrub 50K	0.05	0.04	0.05	0.05	0.02	0.02
Conservative	0	0	0	0	0	0

Table 4: Vulnerability evaluation - Each row shows object bytes (normalized to Pmemobj) accessed without checksum verification.

The average modified sizes ("Mod" rows) determine the logging size and affect the performance drop between Pangolin and Pangolin-ML. Note that a transaction does not necessarily modify (and log) the whole range of an involved object. The performance difference between Pangolin-ML and Pangolin-MLP is a consequence of both allocated and modified sizes.

For insert transactions, Pangolin is faster than Pmemobj for ctree and btree, but slower than Pmemobj for other data structures. This is because the slower applications have relatively small modified sizes compared to the object sizes, and Pangolin's data movement from NVMM to micro-buffers overshadows its advantage for whole-object updates, as shown in Figure 3. For remove transactions, Pangolin is marginally faster than Pmemobj except for the case of skiplist, which is also because of the data movement caused by micro-buffering.

Pangolin-MLP's performance is 95% of Pmemobj-R on average, and it saves orders of magnitude NVMM space by using parity data as redundancy. Pangolin-MLPC adds scribble detection and performance drops by between 1.5% to 15% relative to Pangolin-MLP. Adding object checksums impacts rtree's transactions the most because the allocated object size is large, which requires more checksum computing time.

Pangolin does not impact the lookup performance because it performs direct NVMM reads without constantly verifying object checksums. Pangolin ensures data integrity with its checksum verification policy, as discussed in Section 3.3.

Figure 6 illustrates the impact of different strategies for checksum verification. We compare Pangolin's default mode (Pangolin-MLPC) with two "Scrub" modes and a "Conservative" mode. The default mode only verifies checksums for micro-buffered objects. In "Scrub" mode, a scrubbing thread verifies data integrity of the whole object pool when a preset number (indicated by legends in Figure 6) of transactions have completed. The "Conservative" mode verifies the checksum for every object access (including those read by `pgl_get` without micro-buffering).

Table 4 quantifies the vulnerability using the amount of object data that is accessed without checksum verification. The data accumulates across all transactions for Pmemobj, Pangolin-MLPC, and "Conservative" modes. For "Scrub" modes, we count the vulnerable data between two scrubbing runs. Numbers in Table 4 are normalized to Pmemobj, which does not have any checksum protection for object data.

The cost of verifying checksums for every object access depends strongly on the data structure size and its insertion algorithm. For small objects, such as ctree, rbtree, and hashmap, the cost is negligible. But for btree, skiplist, and rtree, due to their large object sizes, the cost is significant. Thus, a scrubbing-based policy could be faster, with more data subject

to corruption between two successive runs.

4.6 Error Detection and Correction

Pangolin provides error injection functions to emulate both hardware-uncorrectable NVMM media errors and hardware-undetectable scribbles.

We initially developed Pangolin using conventional DRAM machines that lack support for injecting NVMM errors at the hardware level. Therefore, we use `mprotect()` and `SIGSEGV` to emulate NVMM media errors and `SIGBUS`. When an NVMM file is DAX-mapped, the injector can randomly choose a page that contains user-allocated objects, erase it, and call `mprotect(PROT_NONE)` on the page. Later, when the application reads the corrupted page, Pangolin intercepts `SIGSEGV`, changes the page to read/write mode, and restores the page's data. The injector function can also randomly corrupt a metadata region or a victim object to emulate software-induced, scribble errors.

In both test cases, we observe Pangolin can successfully repair a victim page or an object and resume normal program execution. In our evaluation using a 100 GB pool and 1 GB parity, we measured 180 μs to repair a page of a page column.

We also intentionally introduce buffer overrun bugs in our applications and observe that Pangolin can successfully detect them using micro-buffer canaries. The transaction then aborts to prevent any NVMM corruption. We have also verified Pangolin is compatible with AddressSanitizer for detecting buffer overrun bugs (when updating a micro-buffered object exceeds its buffer's boundary), if both Pangolin and its application code are compiled with support.

5 Related Work

In this section, we place Pangolin in context relative to previous projects that have explored how to use NVMM effectively.

Transaction Support All previous libraries for using NVMMs to build complex objects rely on transactions for crash consistency. Although we built Pangolin on `libpmemobj`, its techniques could be applied to another persistent object system. NV-Heaps [3], Atlas [1], DCT [22], and `libpmemobj` [39] provide undo logging for applications to snapshot persistent objects before making in-place updates. Mnemosyne [47], SoftWrAp [11], and DUDETM [25] use variations of redo logging. REWIND [2] implements both undo and redo logging for fine-grained, high-concurrent transactions. Log-structured NVMM [12] makes changes to objects via append-only logs, and it does not require extra logging for consistency. Romulus [5] uses a main-back mechanism to implement efficient redo log-based transactions.

None of these systems provide fault tolerance for NVMM errors. We believe they can adopt Pangolin's parity and checksum design to improve their resilience to NVMM errors at low storage overhead. In Section 3.5 we described how to apply the hybrid parity updating scheme to an undo logging-based system. Log-structured and copy-on-write systems can adopt the techniques in similar ways.

Fault Tolerance Both Pangolin and `libpmemobj`'s replication mode protect against media errors, but Pangolin provides stronger protection and much lower space overhead. Furthermore, `libpmemobj` can only repair media errors offline, and it does not detect or repair software corruption to user objects.

NVMalloc [32] uses checksums to protect metadata. It does not specify whether application data is also checksum-protected, and it does not provide any form of redundancy to repair the corruption. NVMalloc uses `mprotect()` to protect NVMM pages while they are not mapped for writing. Pangolin could adopt this technique to prevent an application from scribbling its own persistent data structures.

The NOVA file system [48, 49] uses parity-based protection for file data. However, it must disable these features for NVMM pages that are DAX-mapped for writing in user-space, since the page's contents can change without the file system's knowledge, making it impossible for NOVA to keep the parity information consistent if an application modifies DAX-mapped data. As a result, Pangolin's and NOVA's fault tolerance mechanisms are complementary.

6 Conclusion

This work presents Pangolin, a fault-tolerant, DAX-mapped NVMM programming library for applications to build complex data structures in NVMM. Pangolin uses a novel, space-efficient layout of data and parity to protect arbitrary-sized NVMM objects combined with per-object checksums to detect corruption. To maintain high performance, Pangolin uses micro-buffering, carefully-chosen parity and checksum updating algorithms. As a result, Pangolin provides stronger protection, better availability, and much lower storage overhead than existing NVMM programming libraries.

Acknowledgments

This work was supported in part by Toshiba, and we are thankful for their continued sponsorship. We thank our shepherd, Kimberly Keeton and the anonymous USENIX ATC reviewers for their insightful comments and suggestions. We are also thankful for Intel to provide us early access to platforms with Intel's Optane DC Persistent Memory Modules.

References

[1] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM*

International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'14, pages 433–452. ACM, 2014.

[2] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. REWIND: Recovery Write-ahead System for In-Memory Non-volatile Data-Structures. *Proceedings of the VLDB Endowment*, 8:497–508, 2015.

[3] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 105–118. ACM, 2011.

[4] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, SOSP'09, pages 133–146. ACM, 2009.

[5] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA'18, pages 271–282. ACM, 2018.

[6] Dan Williams. libnvdimm for 4.12, 2017. https://lkml.org/lkml/2017/5/5/620.

[7] Dan Williams. libnvdimm for 4.13, 2017. https://lkml.org/lkml/2017/7/6/843.

[8] Dan Williams. use memcpy_mcsafe() for copy_to_iter(), 2018. https://lkml.org/lkml/2018/5/1/708.

[9] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'16, pages 125–136. ACM, 2016.

[10] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[11] Ellis R Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, MSST'16, pages 1–14. IEEE, 2015.

[12] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-Structured Non-Volatile Main Memory. In *Proceedings of the USENIX Annual Technical Conference*, ATC'17, pages 703–717. USENIX Association, 2017.

[13] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 111–122. ACM, 2012.

[14] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/en-us/isa-extensions.

[15] Intel. Introduction to Programming with Persistent Memory from Intel, 2017. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent\-memory-from-intel.

[16] Intel. Persistent Memory Programming - Frequently Asked Questions, 2017. https://software.intel.com/en-us/articles/persistent-memory-programming-\frequently-asked-questions.

[17] Intel. Discover Persistent Memory Programming Errors with Pmemcheck, 2018. https://software.intel.com/en-us/articles/discover-\persistent-memory-programming-errors-with-\pmemcheck.

[18] Intel. Intelligent Storage Acceleration Library, 2018. https://software.intel.com/en-us/storage/isa-l.

[19] Intel. Intel® Optane™ DC persistent memory, 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 427–442. ACM, 2016.

[21] 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. https://arxiv.org/abs/1903.05714.

[22] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 399–411. ACM, 2016.

[23] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies*, FAST'17, pages 197–212. USENIX Association, 2017.

[24] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 27:1–27:14. ACM, 2014.

[25] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 329–343. ACM, 2017.

[26] Tony Luck. Patchwork mm/hwpoison: Clear PRESENT Bit for Kernel 1:1 Mappings of Poison Pages, 2017. https://patchwork.kernel.org/patch/9793701.

[27] LWN. Add Support for NV-DIMMs to Ext4, 2014. https://lwn.net/Articles/613384.

[28] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys'17, pages 499–512. ACM, 2017.

[29] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'15, pages 177–190. ACM, 2015.

[30] Micron. 3D XPoint Technology, 2017. http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[31] Micron. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility, 2017. http://www.micron.com/products/dram-modules/nvdimm.

[32] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS'13. ACM, 2013.

[33] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 135–148. ACM, 2017.

[34] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 401–410. ACM, 2012.

[35] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR'16, pages 7:1–7:11. ACM, 2016.

[36] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100. ACM, 2007.

[37] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, HPCA'18, pages 336–349. IEEE, 2018.

[38] Christian Perone and David Murray. pynvm: Non-volatile memory for Python, 2017. https://github.com/pmem/pynvm.

[39] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[40] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'15, pages 672–685. IEEE, 2015.

[41] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, volume 6, pages 9:1–9:23, New York, NY, USA, September 2010. ACM.

[42] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'09, pages 193–204. ACM, 2009.

[43] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam H Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 91–104. ACM, 2017.

[44] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, ATC'12. USENIX Association, 2012.

[45] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 297–310. ACM, 2015.

[46] UEFI Forum. Advanced configuration and power interface specification, 2017. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 91–104. ACM, 2011.

[48] 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'16, pages 323–338. USENIX Association, 2016.

[49] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 478–496. ACM, 2017.

[50] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.

[51] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. End-to-end Data Integrity for File Systems: A ZFS Case Study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10. USENIX Association, 2010.