# Corundum: Statically-Enforced Persistent Memory Safety

Morteza Hoseinzadeh
University of California, San Diego
San Diego, California, USA
mhoseinzadeh@cs.ucsd.edu

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

**ACM Reference Format:**
Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21), April 19–23, 2021, Virtual, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3445814.3446710

## 1 MOTIVATION

Fast, byte-addressable, persistent main memory (PM) makes it possible to build complex data structures that can survive system failures. PM offers numerous potential benefits including improved memory system capacity, lower-latency and higher-bandwidth storage, and a unified programming model for persistent and volatile program state. However, it also poses a host of novel challenges. For instance, it requires memory controller and ISA support, new operating systems facilities, and it places large, novel burdens on programmers.

PM programming combines well-known programming challenges like locking, memory management, and pointer safety with novel PM-specific bug types. It also requires logging updates to PM to facilitate recovery after a crash. A misstep in any of these areas can corrupt data, leak resources, prevent successful recovery after a crash. Existing PM libraries in a variety of languages – C, C++, Go, Java – simplify some of these problems [2–4, 7, 8], but they still require the programmer to learn (and flawlessly apply) complex rules to ensure correctness. Opportunities for data-destroying bugs abound.

The challenges of programming *correctly* with PM are among the largest potential obstacles to wide-spread adoption of PM and our ability to fully exploit its capabilities. If programmers cannot reliably write and modify code that correctly and safely modifies persistent data structures, PM will be hobbled as a storage technology.

Some of the bugs that PM programs suffer from have been the subject of years of research and practical tool building. The solutions and approaches to these problems range from programming disciplines to improved library support to debugging tools to programming language facilities.

Given the enhanced importance of memory and concurrency errors in PM programming, it makes sense to adopt the most effective and reliable mechanisms available for avoiding them.

## 2 LIMITATIONS OF THE STATE OF THE ART

Existing PM programming libraries like PMDK [7], provide the capability to avoid the errors described above, but they rely on the programmer correctly apply those capabilities. Identifying errors relies to regression tests, stress tests, and code review, all of which have serious drawbacks, especially for subtle coding errors that can lead to non-deterministic, rarely exercised, and hard-to-reproduce bugs.

Recently developed languages like Rust [1] have shown that static analysis can detect and prevent many types of complex, subtle bugs (e.g., locking and memory allocation). However, these languages do not include specific support for PM.

## 3 KEY INSIGHTS

Our work relies on two key insights: First, nearly all PM-related safety properties are amenable to *static analysis and checking*. In particular, PM pointer and memory allocation safety are similar in many respects to their conventional, volatile memory counterparts.

Rust's type system enforces safety invariants for volatile memory and the same facilities can be adapted to statically enforce PM safety invariants as well. The result should be less testing, fewer bugs, and faster code, since the system can avoid dynamic checks in most instances.

## 4 MAIN ARTIFACTS

Corundum[1] [6] is a Rust-based library (or "crate") with an idiomatic PM programming interface and leverages Rust's type system to statically avoid most common PM programming bugs. Corundum lets programmers develop persistent data structures using familiar Rust constructs and have confidence that they maintain important PM safety properties (e.g., persistent pointer safety, race-freedom, and non-leaking memory allocation). Table 2 compares an example of insert operation to a sorted linked-list written in both Rust and Corundum to show the idiomaticity.

## 5 KEY RESULTS AND CONTRIBUTIONS

We have implemented Corundum and found its performance to be as good or better than Intel's widely-used PMDK library. Corundum enforces five invariants on PM programs:

(1) PM pools only contain data that can be safely persistent.
(2) Pointers within a pool are always valid. Pointers between pools or from persistent memory to volatile memory are not possible. Pointers from volatile memory into a pool are safe. Closing a pool does not result in unsafe pointers.

[1]The paper is published in the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2021. It is available in the ACM Digital Library.

| System | | Ptrs-Are-Safe | | | No-Races | Tx-Are-Atomic | | No-Leaks |
|---|---|---|---|---|---|---|---|---|
| | Only-P-Objects | Interpool | NV-to-V | V-to-NV | | Atomicity | Isolation | |
| NV-Heaps [4] | M | D | S | M | S | S | M | RC |
| Mnemosyne [8] | M | D | S | M | S | S | M | M |
| libpmemobj [7] | M | D | M | M | M | M | M | M |
| libpmemobj++ [7] | M | D | M | M | M | S | M | M |
| NVM Direct [2] | D | D | S | D | M | S/M | S/M | M |
| Atlas [3] | M | M | M | M | M | S | M | GC |
| go-pmem [5] | M | M | M | M | M | S | M | GC |
| Corundum | S | S/D | S | D | S | S | S | RC |

**Table 1: Corundum more static checks than other PMEM libraries, using them to meet most of its design goals. ('S'=Static, 'D'=Dynamic, 'M'=Manual, 'GC'=Garbage Collection, 'RC'=Reference Counting)**

```
use std::rc::*;
use std::cell::*;
struct Node { val: i32,
  nxt: Rc<RefCell<Option<Node>>>
}
fn insert(&self, val: i32) {

  let mut nxt = self.nxt.borrow_mut();
  if let Some(n) = &*nxt {
    if n.val > val {
      *nxt = Some(Node { val,
        nxt: self.nxt.clone()
      })
    } else { n.insert(val); }
  } else {
    *nxt = Some(Node { val,
      nxt: Rc::new(RefCell::new(None))
    })
  }

}
```

```
use crndm::default::*;

struct Node { val: i32,
  nxt: Prc<PRefCell<Option<Node>>>
}
fn insert(&self, val: i32) {
  transaction(|j| {
    let mut nxt = self.nxt.borrow_mut(j);
    if let Some(n) = &*nxt {
      if n.val > val {
        *nxt = Some(Node { val,
          nxt: self.nxt.pclone(j)
        })
      } else { n.insert(val); }
    } else {
      *nxt = Some(Node { val,
        nxt:Prc::new(PRefCell::new(None,j),j)
      })
    }
  }).unwrap()
}
```

**Table 2: Comparing the implementation of insertion in a sorted linked-list in Rust and Corundum**

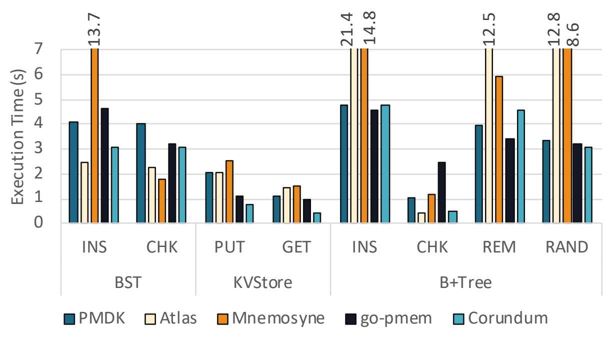

**Figure 1: Performance comparison between PMDK, Atlas, Mnemosyne, go-pmem, and Corundum**

(3) Transactions are atomic with respect to both persistent and volatile data. It is not possible to modify persistent data without logging it.
(4) There are no data races or unsynchronized access to shared persistent data.
(5) Transactions provide isolation so that updates are not visible until the transaction commits.

In almost every case, the compiler can statically detect violations of these invariants. Table 1 compares Corundum's approach to preventing errors with other PM libraries. Corundum has much wider static coverage than any existing system.

We compare Corundum's performance to a selection of well-known PM libraries in Figure 1. The data show that Corundum is as fast as or faster than PMDK while still providing stronger safety guarantees.

## REFERENCES

[1] Rust programming language. https://www.rust-lang.org/.
[2] B. Bridge. NVM Support for C Applications, 2015. Available at http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf.
[3] D. R. Chakrabarti, H.-J. Boehm, and K. 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, New York, NY, USA, 2014. ACM.
[4] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. 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, New York, NY, USA, 2011. ACM.
[5] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872, 2020.
[6] M. Hoseinzadeh and S. Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '21)*, 2021.
[7] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.
[8] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.