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

ABSTRACT

Fast, byte-addressable, persistent main memories (PM) make it possible to build complex data structures that can survive system failures. Programming for PM is challenging, not least because it 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, or prevent successful recovery after a crash. Existing PM libraries in a variety of languages – C, C++, Java, Go – simplify some of these problems, but they still require the programmer to learn (and flawlessly apply) complex rules to ensure correctness. Opportunities for data-destroying bugs abound.

This paper presents Corundum, a Rust-based library 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 will be free of those bugs. We have implemented Corundum and found its performance to be as good or better than Intel's widely-used PMDK library, HP's Atlas, Mnemosyne, and go-pmem.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:
Morteza Hoseinzadeh and Steven Swanson. 2021. Corundum: Statically-Enforced Persistent Memory Safety. In *ASPLOS '21: ACM Symposium on Architectural Support for Programming Languages and Operating Systems, April 19–23, 2021, Virtual.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Persistent main memory (PM) is the first new memory technology to arrive in the memory hierarchy since the appearance of DRAM in the early 1970's. PM offers numerous potential benefits including improved memory system capacity, lower-latency and higher-bandwidth relative to disk-based 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 system facilities, and it places large, new burdens on programmers.

The programming challenges it poses are daunting and stem directly from its non-volatility, high-performance, and direct connection to the processor's memory bus. PM's raw performance demands the removal of system software from the common-case access path, its non-volatility requires that (if it is to be used as storage) updates must be robust in the face of system failures, and its memory-like interface forces application software to deal directly with issues like fault tolerance and error recovery rather than relying on layers of system software.

In addition, programming with PM exacerbates the impact of existing types of bugs and introduces novel classes of programming errors. Common errors like memory leaks, dangling pointers, concurrency bugs, and data structure corruption have permanent effects (rather than dissipating on restart). New errors are also possible: A programmer might forget to log an update to a persistent structure or create a pointer from a persistent data structure to volatile memory. The former error may manifest during recovery while the latter is inherently unsafe since, after restart, the pointer to volatile memory is meaningless and dereferencing it will result in (at best) an exception.

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.

The Rust programming language provides programming language-based mechanisms to avoid a host of common memory and concurrency errors. Its type system, standard library, and "borrow checker" allow the Rust compiler to statically prevent data races, synchronization errors, and most memory allocation errors. Further, the performance of the resulting machine code is comperable with that of compiled C or C++. In addition to these built-in static checks, Rust also provides facilities that make it easy (and idiomatic) to create new types of smart pointers that integrate cleanly with the

ASPLOS '21, April 19–23, 2021, Virtual

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

rest of the language. Its type systems also make data modification explicit and easy to control.

We have leveraged Rust's abilities to create Corundum, a library (or "crate" in Rust parlance) that lets programmers build persistent data structures. Corundum provides basic PM programming facilities (e.g., opening and mapping persistent memory pools) and a persistent software transactional memory interface to provide atomic updates to persistent data.

Uniquely, Corundum uses (mostly) static checking to enforce several key PM programming invariants:

- Corundum prevents the creation of unsafe pointers between non-volatile memory regions (or "pools") and pointers from those pools into volatile memory.
- Corundum ensures that programs only modify persistent memory within transactions and that they log all updates to persistent state.
- Corundum prevents most persistent memory allocation errors (e.g. dangling pointers, and multiple frees) in the presence of multiple, independent pools of PM.

Our experience building Corundum demonstrates the benefits that a strong type system can bring to PM programming. We evaluate Corundum quantitatively and qualitatively by comparing it to existing PM programming libraries. We find that Corundum provides stronger guarantees than other libraries and that it is as fast or faster than most of them. We also highlight some changes to Rust that would make Corundum even more powerful and efficient.

The rest of paper is organized as follows. Section 2 gives a brief background information on PM programming and the Rust language. Section 3 describes Corundum. In Section 4 we evaluate our library. Section 5 places Corundum in context relative to related work. Finally, Section 6 concludes.

2 BACKGROUND

Corundum adds PM programming support to Rust by providing facilities for accessing persistent memory, providing a transaction mechanism to ensure consistency in the case of failure, and statically detecting common PM programming bugs.

To accomplish this, Corundum confronts a set of challenges that are common among PM programming systems for languages like C, C++, and Java. Corundum addresses these challenges using the unique features of Rust.

2.1 Persistent Memory Programming

The emergence of persistent main memory technologies (most notably Intel's Optane DIMMs) [27] brings byte-addressable, persistent memory to modern processors. The persistent memory appears in the processor's physical address space.

The most popular operating system mechanism for exposing persistent memory to applications is to use a PM-aware file system to manage a large region of persistent memory. An application can use a direct access (DAX) `mmap()` system call to map a file in the file system into its virtual address space. From there, the application can access the memory directly using load and store instructions, avoiding all operating system overheads in the common case.

2.1.1 PM Programming Support. While DAX `mmap()` provides access, productively and safely using PM presents challenges that typical PM programming libraries address. To be successful, Corundum must address these as well. Below, we outline the most important of these challenges.

PM libraries provide access to multiple independent *pools* of persistent memory with a *root* object from which other objects are reachable.

Each pool has a private, atomic memory allocator for allocating and reclaiming space within the pool that is robust in the face of system crashes.

Libraries usually identify which types can exist in a pool. For instance, the library may provide a base class or interface that persistent object should inherit from or implement.

Finally, PM libraries provide a means to express atomic sections, usually in the form of a persistent software transactional memory [8, 11, 33] mechanism that specifies regions of code that should be atomic both with respect to failure and to concurrent transactions.

2.1.2 PM Bugs. PM programmers must reckon with a wide range of potential bugs that can cause permanent corruption, lead to unsafe behavior, or leak resources. These include common memory allocation/deallocation errors and race conditions as well as PM-specific challenges.

The three most critical types of PM-specific bugs are logging errors, unsafe inter-pool pointers, and pointers into closed pools.

Logging errors An atomic section must log all persistent updates so the transaction can roll back in case of a system failure [9–11, 13, 28, 36]. Failing to manually log an update (as some systems require) can let a poorly-timed system crash compromise data integrity.

Updates can be hard to recognize in code, leading to unlogged updates. For instance, passing the address of a field of a persistent struct to a library function might (to the programmer's surprise) modify the data it points to.

Inter-pool pointers PM programs can simultaneously access several, independent PM pools in addition to the conventional, volatile heap and stack. The PM pools need to be self-contained so that one pool does not contain a pointer into another pool or into volatile memory. Dereferencing such a pointer is certain to be unsafe after a restart.

Pool Closure If a pool closes, the system must unmap the memory it contains. This leaves any pointers from DRAM into the pool unsafe [11].

2.2 Rust

The Rust programming language [19] is intended for parallel, low-level, systems programming and it uses a sophisticated type system to prevent a range of common programming errors. For instance, its type system makes data races impossible and it provides cheap reference counting garbage collection. The result is a thoroughly modern language that is about as fast as C or C++ but with vastly stronger safety guarantees.

Rust is a well-loved by the developers who use it [35] despite having a moderately steep learning curve due to it requiring programmers to think differently about variable lifetimes, mutability, and concurrency.

Since the memory allocation, synchronization, and safety invariants that PM programming provides are a strict superset of those for conventional volatile programming, we can leverage Rust's safety guarantees to improve the safety of PM programming.

Many guarantees that Rust provides are not built in the language itself, but are implemented in its standard library. This means that we can use the same language features to enforce new guarantees in an idiomatic and familiar (to Rust programmers) way.

Below we, describe the key features of Rust that Corundum uses to provide PM-specific safety guarantees. For a thorough introduction, consult the Rust manual [19].

2.2.1 Mutability, Immutabilty, and Interior Mutability. Mutability is a central concept in Rust that governs when a value may change. The critical property that Rust maintains is that there can be one mutable reference to a piece of data or multiple immutable references, but not both. We refer to this as *the mutability invariant.* In most instances, Rust enforces this invariant statically.

In some cases, static checks are too restrictive or the program may need to enforce further constraints on when data can be mutable. Rust provides *interior mutability* for these situations, and the wrapper type `RefCell` exemplifies this concept. Curiously, `RefCell`s are always immutable so assigning to them directly is impossible. However, the program can acquire mutable and immutable references by calling `RefCell::borrow_mut()` and `RefCell::borrow()`, respectively. These functions return reference objects (similar to smart pointers) that programs can use to access the data. The key is that the `RefCell` dynamically enforces the mutability invariant: Calling `borrow()` when a mutable reference object exists or `borrow_mut()` when any reference object exists will cause a `panic!()`.

2.2.2 Smart Pointers, Wrappers, and Dynamic Memory Allocation. Rust uses smart pointers and type wrappers to implement memory management, garbage collection, and concurrency control. Defining new kinds of smart pointers, type wrappers, and guard objects that integrate cleanly into the language is easy, since Rust makes dereferencing smart pointers fully transparent.

Rust's standard library provides a range of smart pointer types and type wrappers (the <> notation is Rust's syntax for generics):

(1) `Box<T>` is a pointer to an object of type `T` allocated on a the heap. When a `Box<T>` goes out of scope, the allocated data is freed.
(2) `Rc<T>` is a reference-counted pointer to a heap-allocated object of type `T`. Multiple instances of `Rc<T>` can point to the same data, and when the last `Rc<T>` goes out of scope, the data is reclaimed. Since multiple `Rc` instances can refer to the same data, `Rc<T>` does not allow the modification of data it holds.
(3) `RefCell<T>` (described above) is a wrapper type holds an object of type `T` that is immutable except via interior mutability.
(4) `Mutex<T>` is a thread-safe version of `RefCell`. Its `lock()` method, locks the mutex and returns a reference object that is also a guard object for the mutex. When the reference goes out of scope, the lock is released.

Each of these wrappers encapsulates a specific set of capabilities, and programmers can compose them to build feature-rich data types. For instance, a common idiom – `Rc<RefCell<T>>` – combines `RefCell` with `Rc` to provide shared, mutable objects.

When the last reference to an object dies, Rust "drops" it. Dropping a struct recursively drops any fields it contains and types can implement `drop()` to implement destructor-like functionality. For instance, `drop()` for `Box` deallocates the memory it holds, and `drop()` for `Rc` decrements the reference count and frees the memory if it reaches zero.

2.2.3 Option Types. Rust provides optional values in form of `Option<T>` that can be either `Some(data)` (where `data` is of type `T`), or `None`. Its most common use to allow for null pointers: `Box<T>` is always a valid pointer to allocated memory, but `Option<Box<T>>` can either be `None` or a `Box<T>`.

`Option` is one instance of Rust's `enum` mechanism, and Rust provides a `match` construct that is similar to `switch` in C, but requires the programmer provide code for all possible values. For `Option<PBox<T>>`, this avoids null dereferences and forces code to account for pointers that might be null.

2.2.4 Traits and Type Bounds. Rust provides *traits* that are similar to Java or C# interfaces. Structures can implement traits. A ubiquitous Rust idiom is to "bound" or restrict the types that can be used in a particular context based on the traits they implement.

Of particular importance are "auto traits" that all types (even primitive types) implement by default but can opt out of. For instance, all types implement the `Send` trait so they can be sent to another thread. Rust allows negative implementation (using `!` symbol) of auto traits for opting out of it. For instance, `Rc<T>` is labeled as `!Send` because it uses non-atomic counters and is, therefore, not safe to share among threads.

2.2.5 Panic. In response to serious errors, programs can call `panic!()` to end the program. Rust provides an exception-like mechanism to catch panics and attempt to recover from them.

2.2.6 Unsafe Rust. Rust provides `unsafe` code blocks which give the programmer access to a superset of normal, safe Rust. Unsafe Rust allows C-like direct manipulation of memory, unsafe casts, etc. The Rust standard library uses unsafe Rust to implement interior mutability and many other critical features (e.g., system calls and low-level memory management).

Libraries like Corundum can declare types or functions to be unsafe, preventing their invocation or use in safe code.

If programmers decide to use `unsafe` constructs, they assume responsibility for enforcing the guarantees that Rust (or the library) relies upon. Typical Rust programmers do not use `unsafe` constructs.

3 CORUNDUM

Corundum is a Rust library (or "crate") that uses static and dynamic checking to avoid common PM programming errors. Aside from providing these strong safety guarantees, Corundum is similar to other PM programming libraries like PMDK [33], NV Heaps [11], and Mnemosyne [36]. Corundum provides four abstractions – typed persistent memory pools, transactions, persistent smart pointers, and atomic memory allocation – that address the challenges and common bugs described in Section 2.1.

Corundum strives to provide strong safety guarantees for persistent memory programming. It achieves the following design goals:

Design Goal 1 (Only-Persistent-Objects)**:** *Pools only contain data that can be safely persistent.*

Design Goal 2 (Ptrs-Are-Safe)**:** *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.*

Design Goal 3 (Tx-Are-Atomic)**:** *Transactions are atomic with respect to both persistent and volatile data. It is not possible to modify persistent data without logging it.*

Design Goal 4 (No-Races)**:** *There are no data races or unsynchronized access to shared persistent data.*

Design Goal 5 (Tx-Are-Isolated)**:** *Transactions provide isolation so that updates are not visible until the transaction commits.*

Design Goal 6 (No-Acyclic-Leaks)**:** *Memory leaks in acyclic data structures are not possible.*

Corundum achieves these goals through a combination of several core techniques that are embodied by Corundum's abstractions. Corundum borrows and relies upon existing features of Rust to prevent data races and avoid unsafe pointer manipulation. These properties are a foundation that Corundum builds upon.

Corundum breaks a program's execution into transactions and non-transactional code, and a thread's execution alternates between the two. Transactions are atomic, isolated, and can modify persistent state but cannot modify pre-existing volatile state. The remaining, non-transactional code can modify volatile state but cannot modify persistent state.

Critically, Corundum limits the kinds of variables that can cross transaction boundaries, and this allows for careful reasoning about the invariants that hold at the boundaries. For instance, Corundum guarantees that leaks are not possible by showing that they do not exist at transaction boundaries.

3.1 Assumptions

The correctness of Corundum rests on several assumptions. We assume that the program does not use `unsafe` constructs, since unsafe Rust can bypass almost all of the guarantees that Rust makes and relies upon.

We also assume that our implementation of the Corundum memory allocator, logging and recovery code, and reference counting system are correct. We have tested them thoroughly and techniques for building correct versions of these components are well-known [8, 9, 11, 33, 36].

Finally, students of Rust will notice that we do not discuss some common types (e.g., `Cell` or `Weak`) or their persistent analogs. Corundum's treatment of these types is analogous to the types we discuss and Corundum provides persistent counterparts. We have omitted the details for brevity.

3.2 Corundum Pools and Objects

Corundum provides self-contained pools of persistent memory and constrains the data types that they can hold.

Pools Corundum pools reside in a PM-backed file. The pool contains some metadata, a root pointer, persistent memory allocation data structures, and a region of persistent memory.

Corundum identifies each memory pool with a *pool type*, and all persistent types take a pool type as a parameter. This statically binds each persistent object to its pool. Likewise, transactions are bound to a particular pool via the pool type. In our discussion, we use `P` as a representative pool type.

Programs open a pool by calling `P`'s `open()` function: `P::open<T>(foo)`. This binds `P` to the pool in file `foo`. Corundum ensures that only one open pool is bound to `P` at any time. The type parameter, `T`, determines the type of the root pointer. `P::open<T>()` returns an immutable reference to the root pointer. Pools remain open until reference to the root object drops.

Programmers can statically declare multiple pool types using the `pool!()` macro, but the number of pool types available (and, therefore, the number of simultaneously open pools) is fixed at compile time.

Persistent Objects Objects in a pool must implement the `PSafe` auto trait. Corundum declares all primitive arithmetic types to be `PSafe`, so these types and structs composed of them are `PSafe`.

`PSafe` (like all of Corundum's auto traits) is declared `unsafe`, so programmers should not explicitly label types as `PSafe` or `!PSafe`.

Reference and smart pointer types that point to volatile data (e.g., references, mutable reference, `Box`, `Arc`, `Rc`, and `Mutex`) and types that refer to state external to the program (e.g., file handles) are `!PSafe`.

3.3 Transactions: The Basics

All modifications to a Corundum pool occur within a transaction, including memory allocations and modifications of the root pointer.

Programs create transactions by passing an anonymous function (i.e., a lambda) to `P::transaction()` (Lines 19–21 of Listing 1). The lambda takes a single argument, `j`, which will be reference to a *journal object* that holds information about the current transaction. Variable `j` is of type `Journal<P>`, so it is bound to pool `P`.

The lambda can capture values from the transaction's lexical scope (e.g., `head` in Listing 1), allowing transactions to integrate smoothly into the surrounding code. `P::transaction()` returns the return value of the transaction body.

Corundum flattens nested transactions: Modifications to a pool commit when outermost transaction for that pool commits.

The programmer is responsible for acquiring locks in the correct order to avoid deadlock.

Rust's type system allows Corundum to restrict the inputs and output of the transaction. The `TxInSafe` auto trait bounds the types a transaction can capture. Corundum marks all volatile mutable references, smart pointers, wrappers, and interior mutability types as `!TxInSafe`.

This provides our first invariant:

Invariant 1 (TX-No-Volatile-Mutability)**:** *Transactions cannot modify existing volatile state.*

```
#[Root]
struct Node {
  val: i32,
  next: PRefCell<Option<Pbox<Node,P>>,P>
}
fn append(n: &Node, v:i32, j: &Journal<P>) {
  let mut t = n.next.borrow_mut(j);
  match &*t {
    Some(succ) => append(succ, v, j);
    None => *t = Some(Pbox::new(
        Node {
          val: v,
          next: PRefCell::new(None, j)
        }, j));
  }
}
fn go(v: i32) {
  let head = P::open::<Node>("list.pool",0);
  P::transaction(|j| {
    append(&head, v, j);
  });
}
```

Listing 1: A Corundum implementation of linked list append. Some error management code has been elided for clarity.

```
1  let mut done = false;
2  let p1 = P::transaction(|j|{
3      let p1 = Pbox::new(1, j);
4      let p2 = Pbox::new(2, j);
5      root.set(p2);
6      done = true;
7      ^^^^ the trait `TxInSafe` is not
8           implemented for `&mut bool`
9      p1
10     ^^ the trait bound `Pbox<i32,P>:TxOutSafe`
11        is not satisfied
12 }).unwrap();
13 ^ `p1` is dropped here.
14 ^ `p2` is alive and durable here because it
15   is reachable from the root object.
```

Listing 2: Reachability though lifetime and type bounding

Details: Since all the types that provide mutable access to volatile data are `!TxInSafe`, pre-existing instances of these variables are not available inside the transaction (Line 6 of Listing 2). However, variables can be created and modified within the transaction. Pre-existing volatile data can be read.

```
P1::transaction(|j1| {
    let v = Box::new(10);
    let p1 =
      Pbox::new(v, j1);
                 ^ Box<i32>:PSafe is not satisfied
}).unwrap();
```

Listing 3: Only persistent-safe objects are acceptable.

`TxOutSafe` bounds the values a transaction can return. All references and pointers are `!TxOutSafe`, so transactions can return data only by value. Line 9 of Listing 2 shows how the compiler complains when a user attempts to send out a persistent object.

Corundum also uses the concept of a *stranded type*: If a type is `!TxOutSafe`, `!Send`, and `!PSafe`, then instances of that type cannot escape a transaction. This is because 1) Since a stranded type is `!TxOutSafe`, the transaction cannot return it, 2) since it is `!PSafe`, it cannot be stored in a pool for later retrieval, 3) since it is `!Send`, it cannot be passed to another thread, and 4) it cannot be assigned to a volatile variable (TX-No-Volatile-Mutability).

Design Goal 1 (Only-Persistent-Objects) Holds: The declaration of `P::open()` includes a bound to ensure that the root is `PSafe`. If it contains references or pointers they must be one of the persistent reference/pointer types (see below), since volatile pointers and references are `!PSafe` (e.g. Listing 3).

`PSafe` is a type bound on all of the persistent smart pointer and wrapper types, so those types can only point to, refer to, or wrap `PSafe` objects.

Corundum also carefully constrains the availability of journal objects:

Invariant 2 (TX-Journal-Only)**:** *Journal objects are only available inside transactions.*

Details: The constructor for the `Journal<P>` is unsafe, so the program cannot safely create one. Therefore, the only journal objects that might be available are the ones passed to a transaction as an argument. `Journal<P>` is declared to be stranded, so it cannot escape the transaction.

3.4 Pointers to Persistent Data

Corundum's smart pointer, smart reference, and wrapper types play a crucial role in avoiding persistent programming errors, since they mediate access to persistent state. Their design prevents pointers from one persistent pool to another, prevents the modification of persistent state outside of transactions, and plays a role in avoiding memory leaks.

Table 1 summarizes Corundum's persistent smart pointer types. With the exception of `VWeak`, they mirror Rust's volatile smart pointers (See Section 2). The interface differs in two ways: First, each type takes a pool type as a type parameter, so pointers that reside in different pools have different types. Second, their constructors and mutating accessors take a journal object as an argument.

Three of the methods listed – `PRefCell::borrow()`, `PRefCell::borrow_mut()`, and `PMutex::lock()` – return objects that behave like references. These objects are all stranded.

Corundum type	API	Description
PMEM Smart Pointers for Dynamic Allocation		
Pbox<T,P>		Statically scoped, unshared pointer to PMEM. Deallocates when it goes out of scope.
	new(value: T, j: &Journal<P>)	Allocate PMEM in P and initialize it.
Prc<T,P>		Dynamic PMEM allocation with thread-unsafe reference counting.
Parc<T,P>		Dynamic PMEM with thread-safe reference counting.
	new(value: T, j: &Journal<P>)	Allocate PMEM in P and initialize it.
	pclone(j: &Journal<P>)	Create a new reference to the data.
	downgrade()->PWeak	Return a persistent weak (PWeak<T,P>).
	demote()->VWeak	Return a volatile weak pointer (VWeak<T,P>).
PWeak<T,P>	upgrade(j: &Journal<P>)->Prc/Parc	Convert to a Option<Prc/Parc<T,P>> if it is available
VWeak<T,P>	promote(j: &Journal<P>)->Prc/Parc	Convert to a Option<Prc/Parc<T,P>> if it is available
PMEM Wrappers for Interior Mutability		
PCell<T,P>		Interior mutability via copying data to and from PMEM using get() and set() functions.
PRefCell<T,P>		Interior mutability via references with dynamic borrow checking.
	new(value: T, j: &Journal<P>)	Create new instance on the stack and initialize it.
	borrow()	Return an immutable reference to value it contains.
	burrow_mut(j: &Journal<P>)	Return a mutable reference object (RefMut) for the value inside.
PMutex<T,P>		Thread-safe interior mutability via references.
	new(value: T, j: &Journal<P>)	Create new instance on the stack and initialize it.
	lock(j: &Journal<P>)	Lock and return a mutable reference (PMutexGuard).

Table 1: Corundum's smart pointers and type wrappers for persistent data corresponding closely Rust's pointers and wrappers for volatile data. The key differences are that the Corundum types takes pool type as a type parameter, binding them to a particular pool.

VWeak is a pointer from volatile memory into a pool. It is "weak" in the sense that it does not affect reference counts. The promote() method grants access to the data a VWeak refers to by providing a Parc that refers to the same data. Promoting is only possible within a transaction. Parc::demote() creates VWeak pointers.

Design Goal 2 (Ptrs-Are-Safe) Holds: The presence of multiple pools of PM alongside the volatile heap and stack means the potential for several different kinds of pointers, complicating pointer safety.

Pointers within a pool These pointers are allowed and Corundum relies on Rust's safety properties and mechanisms to ensure their safety.

Pointers between pools Inter-pool pointers are inherently unsafe. They are not possible in Corundum, because pointers to different pools have different types, and assignment between types is not allowed in Rust.

Pointers from a pool to volatile memory These pointers are also inherently unsafe. They are also disallowed: Pools only contain PSafe objects (Only-Persistent-Objects) and pointers to volatile memory are !PSafe.

Pointers from volatile memory into a pool The object a VWeak refers to can disappear if the last Parc refering to the object goes out of scope, deallocating the memory. In this case, promote() will return None, which is safe in Rust.

Pointers into closed heaps If a heap closes, dereferencing any pointers into the heap becomes unsafe. Corundum combines three approaches to prevent this.

First, accessing a pointer to a closed pool from within a transaction is not possible, since P::transaction() will panic!() if P is closed.

Second, consider a reference (other than VMWeak, A, that exists outside a transaction. Rust's borrow checker requires that chain of

```
P1::transaction(|j1| {
    let p1 = LogCell::new(
        Pbox::new(1, j1), j1);
    P2::transaction(|j2| {
        ^^^^^^^^^^^ `j1` is not `TxInSafe`
        let p2 = Pbox::new(1, j2);
        p1.set(p2, j1);
                 ^^ expected P1, found P2
    }).unwrap();
}).unwrap();
```

Listing 4: Cross-Pool referencing prevention via type system

in-scope references exists from the root to A. This includes the root pointer, whose liveness prevents P from closing.

Finally, VWeak pointers from volatile memory into the pool can exist after the pool closes. However, VWeak::promote() requires a journal object to retrieve a usable reference, so it can only be called from inside a transaction, which is only possible if P is open (see above).

3.5 Transactions: Mutability and Isolation

Corundum allows modification of persistent data only via interior mutability and only inside a transaction. Two wrapper types in Table 1 provide interior mutability for persistent data: PMutex and PRefCell.

PMutex::lock() returns a mutable reference to the data while acquiring a lock. The lock is automatically release at the end of the transaction.

PRefCell returns mutable and immutable references via PRefCell::borrow_mut and PRefCell:borrow(), respectively. It dynamically enforces Rust's mutability invariants for these references.

Invariant 3 (Mutable-In-Tx-Only)**:** *Mutable references to persistent data in* `P` *can only exist inside transactions on* `P`*.*

Details: The program can create a mutable reference to persistent data by calling `borrow_mut()` or `lock()` on a smart pointer. Since both of these functions require a journal object as a parameter, they can only be called inside a transaction (TX-Journal-Only).

The resulting mutable reference object (either a `PRefMut` or a `PMutexGuard`) is stranded, so it will be destroyed when it goes out of scope at the end of the transaction.

The reference that `open()` returns to the root object is immutable, so initially, there are no mutable references to pool data available outside a transaction.

Since new mutable references that a transaction creates are stranded, the number of such references outside a transaction cannot increase, so there will never be such a reference.

Design Goal 3 (Tx-Are-Atomic) Holds: For persistent data, Corundum's atomicity guarantee relies on the atomicity of the memory allocator and on all modifications to persistent data being logged.

The allocator and journal object ensure that allocations do not become persistent until the transaction commits. Therefore, on a system failure or `panic!()`, the allocations roll back to reclaim the allocated memory.

Corundum enforces logging by requiring a journal object to make changes to data. To modify persistent data, the program needs a mutable reference object from `PRefCell::borrow_mut()` or `PMutex::lock()`. The reference object performs undo logging the first time it is dereferenced.

Atomicity should include updates to volatile state as well, since a transaction can abort if it calls `panic!()`. Corundum transactions are trivially atomic for volatile state because they cannot modify volatile state (TX-No-Volatile-Mutability).

Design Goal 4 (No-Races) Holds: Rust prevents data races and unsynchronized access using the mutability invariant, the `Mutex` type, marker types to restrict data movement between threads. Corundum takes the same approach by providing `PMutex`, and achieves the same safety guarantees.

Design Goal 5 (Tx-Are-Isolated) Holds: Isolation requires that changes in an uncommitted transaction are not visible to concurrently executing code.

Since transactions cannot modify shared volatile data (TX-No-Volatile-Mutability), we only need to consider changes to persistent objects.

A thread must hold a lock before reading or writing shared persistent state and this can only occur inside a transaction (TX-Journal-Only). Once a thread holds the mutex, no other thread can read the data it protects until the transaction commits and the lock is released, so other threads are isolated from those changes.

3.6 Memory Management

Corundum constrains where programs can allocate and deallocate persistent memory and provides an allocator that can atomically commit or roll back all the allocations and deallocation that occur in a transaction.

Corundum adopts Rust's reference counting garbage collection mechanism. `Parc` and `Prc` smart pointers provide persistent reference counting. Corundum (and Rust) also support weak references to allow for cyclic data structures.

Like Rust's `Rc` and `Arc`, `Prc` and `Parc` provide a `clone()` methods to create a new strong reference to the shared data and increment the reference count. The reference counts are persistent, so modifying them requires `clone()` to take a journal object.

Allocation The only way to allocate persistent memory is by creating `Pbox`, `Prc`, or `Parc` instances. Since the constructors for these types require a journal object, allocation cannot occur outside a transaction.

Deallocation When a reference count goes to zero (for `Prc` or `Parc`) or a `Pbox` goes out of scope, the variable is "dropped" signifying that the allocator can reclaim the memory. However, instead of releasing it immediately, Corundum logs the release and performs it during transaction commit.

Logging occurs in `drop()` (i.e., the destructor) for `Parc` and `Pbox`. Corundum must ensure that deallocation only occurs within a transaction.

This guarantee holds since the destruction of an object only occurs in response to a change in another persistent object (e.g., the destruction of the last reference to that object). Since Corundum only allows changes to persistent memory inside transactions (Mutable-In-Tx-Only), the resulting object destruction will occur in the same transaction.

3.7 Memory Allocation

Corundum extends Rust's reference-counting memory management system prevents memory leaks (in the absence of cycles) and multiple-frees. The main difference between Corundum's memory management scheme and Rust's is that Corundum ensures that newly allocated persistent memory is reachable from some previous allocated persistent memory. This ensures that crashes (which invalidate all volatile pointers) do not leak memory. Corundum uses three mechanisms to enforce this property. First, the atomicity of memory allocations within a transaction prevents the creation of orphaned data if a transaction does not commit. Second, the transaction cannot assign a newly-allocated PMEM region to a captured volatile variable (TX-No-Volatile-Mutability). This prevents persistent data from being kept alive solely by a volatile reference. Third, since the persistent pointers types are `!TxOutSafe`, PMEM data cannot escape the transaction via the transaction's return value.

As a result, the only way a new PMEM allocation can outlive the transaction is to become reachable from a region of persistent memory that was allocated in an earlier transaction.

Design Goal 6 (No-Acyclic-Leaks) Holds:

We begin by dividing `P`'s allocatable space (i.e., excluding per-pool metadata) into *blocks*. Every block is in one of two states: *allocated* or *free*.

A block, B is *reachable* from another block A, if A contains a `Pbox`, `Prc`, or `Parc` that points to B or another block, C such that B is reachable from C. Further assume there are no cycles among the pointer in `P`.

We proceed by induction on the number of transactions executed for P over its entire lifetime, including system failures, recoveries, and restarts. Initially, all of P's blocks are free and there are no orphans.

Since transactions roll back on failure, failed transactions cannot create orphans.

To show that a Corundum transaction that commits does not create an orphan, consider a newly created `Pbox`, `B`, in an orphan-free pool, `P`.

We need to show that either `B` becomes reachable in `P` *or* that all references to `B` go out of scope at the end of transaction, so the memory that `B` references will be reclaimed. That is, we must show that if `B` is not dead at the end of the transaction, it is reachable from the root of `P`.

We can assume `B` is live throughout the transaction, otherwise it will be dropped and its memory reclaimed. Consider two possibilities: First, `B` could be *non-locally live* so that it is reachable via a variable that the transaction captured. Alternately, `B` could be *locally live* and be reachable *only* via variables that are local to the transaction (i.e., not reachable via any non-local reference).

At the end of the transaction, if `B` is *locally live*, then all the references to it will go out of scope when the transaction ends, and Corundum will reclaim `B`'s memory. The only potential escape would be for the transaction to return `B` or a locally live reference to `B` but `B` is `!TxOutSafe`, so that is impossible.

If `B` is non-locally live, the transactions must have made it so by assigning it to a variable, `A`, that is reachable from a captured variable. Only persistent captured variables can be modified in a transaction (TX-No-Volatile-Mutability), so `A` must be persistent.

`A` cannot be an orphan since we assumed (inductively) that there were no orphans before the transaction started. Therefore `A` is reachable from the root and now so is `B`.

The argument above hinges on the invariant that a reference to orphaned memory cannot exist outside a transaction. However, Rust's mechanism for spawning a new thread provides a potential escape. Consider this example:

```
P::transaction(|j| {
  let a = Parc::new(j, 42);
  spawn(|k| {
    let b = a;
  });
});
```

Rust's `spawn` functions executes its argument (a lambda) in a new thread. In the code, `a` is an orphan but the thread captures a reference to it. The thread's body is outside the transaction, so our invariant does not hold. To restore the invariant, Corundum makes `Pbox` and `Parc` `!Send`. This prevents their capture by a thread body.

No-Acyclic-Leaks only holds for acyclic data structures, since `Parc` and `Prc` use simple reference counts and cannot detect cycles. If cycles exist, Corundum can leak memory although no more so than Rust itself can.

3.8 Example

Listing **??** implements `append()` for a persistent linked list in Corundum. A `Node` contains an integer and a link to the next `Node`. The link is of type `PRefCell<Option<Pbox<Node,P>>,P>`, which might seem daunting, but this is typical for a Rust pointer declaration. To break it down: `PBox<Node, P>` is pointer to a `Node` in pool `P`. `Option<>` allows the pointer to be `None`. Wrapping the `Option` in `PRefCell` allows for modification via interior mutability.

The function `append()` recursively finds the end of the list, `n` and adds a `Node`. Line 7 uses `PRefCell::borrow_mut()` to get a mutable reference, `t`, to the `Option` object the `PRefCell` contains. Line 8 uses Rust's `match` construct to safely handle all possible values of `t`: `None` or `Some`. In the `Some` (i.e., non-null) case, it binds the content of the `Option` (which has type `Node`) to `succ`, and recursively calls `append()`.

If the `Option` is `None`, the code has reached the end of the list. Line 10 creates a `PBox` to allocate a new `Node` with value `k` and a `next` pointer equal to `None`. It wraps the `PBox` in a non-null value of type `Option`, and assigns it to the mutable reference.

Function `go()` opens "list.pool" and binds it to pool type `P`. The root pointer will hold a `Node` struct. Line 19 starts a transaction, which provides a journal object, which Line 20 passes to `append()`.

Several aspects of the code are notable. First, `head` and `n` are both immutable, so changes are not possible until `borrow_mut()` uses interior mutability to return a mutable reference object. Second, we must pass `j` into `append()` to ensure it executes in a transaction thereby allowing the call to `borrow_mut()` and the memory allocation (Line 10). Third, although we create call `borrow_mut()` for every link in the list, Corundum only logs the last one, since logging only happens when `*t` dereferences the reference object (Line 10). Forth, as written, `Node` and `append` only work on pool type `P`. A more complete implementation would be make `P` a generic type parameter, so they could work on any pool type.

3.9 Limitations and Potential Improvements to Rust

Corundum's design statically prevents many but not all bugs that might occur in persistent programs. The design decisions of Rust also impact Corundum's design and place some limits on what it can achieve.

Uncaught Bugs Corundum aims to protect the programmer from errors that violate the basic rules of persistent memory programming as enshrined in its design goals. However, Corundum does not attempt to protect against higher level errors (e.g., whether a persistent hash map will function correctly).

Dynamic Checks In some cases, Corundum provides dynamic, rather than static, checks. In these case we could not find a way to enforce them with Rust's type system.

Corundum performs dynamic checks to protect against unsafe dereferencing of `VWeak` pointers from DRAM into PM that can arise when a pool closes. Dereferencing these pointers is common – imagine a volatile index that stores pointers to persistent objects – so static checks would be preferable. An enhanced version Rust's lifetime mechanism might be of use in this case, since it might be

System	Ptrs-Are-Safe				Tx-Are-Atomic			
	Only-P-Objects	Interpool	NV-to-V	V-to-NV	No-Races	Atomicity	Isolation	No-Acyclic-Leaks
NV-Heaps [11]	M	D	S	M	S	S	M	RC
Mnemosyne [36]	M	D	S	M	S	S	M	M
libpmemobj [33]	M	D	M	M	M	M	M	M
libpmemobj++ [33]	M	D	M	M	M	S	M	M
NVM Direct [8]	D	D	S	D	M	S/M	S/M	M
Atlas [9]	M	M	M	M	M	S	M	GC
go-pmem [9]	M	M	M	M	M	S	M	GC
Corundum	S	S/D	S	D	S	S	S	RC

Table 2: 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)

able to keep the pool open until all pointers into it went out of scope.

Threads in Transaction Corundum goes through some contortions to make it safe to spawn a thread inside a transaction, since there is no way to restrict where thread spawning is possible. The challenge is that `thread::spawn()` may leak an unreachable `Parc` when called in a transaction. To prevent this, `Parc` is `!Send`, which requires using `Parc::VWeak` to pass persistent pointers to child threads, which is cumbersome.

A solution would be to generalize Rust's ability to bound the variables that a transaction can capture to include functions. We could then make `thread::spawn() !TxInSafe` to prevent the transaction from calling it.

Deadlock Corundum does not prevent deadlock despite several demonstrations that this is possible in a TM system [6, 15, 32, 38]. We did omitted deadlock detection and recovery for simplicity and to align `PMutex`'s behavior with `Mutex`'s.

Log-Free Programming Corundum is more restrictive than most existing PM libraries. For instance, many high-performance PM data structures are log-free and use carefully-ordered updates to ensure crash consistency. More permissive libraries allow this, but such code would not compile under safe Corundum. Ideally, Corundum could grow to include `unsafe` facilities that allow for log-free programming without completely sacrificing its safety properties.

Cyclic References Like all reference-counting memory management systems, Rust (and Corundum) can leak memory in cyclic data structures. The consequences are more severe for Corundum, since the memory is persistent. Several solutions are possible (e.g., a more general garbage collection mechanism), but they would increase complexity, reduce performance, and create a mismatch between Corundum's behavior and Rust's.

Other Languages We chose Rust as the basis for Corundum after considering several alternatives. C is notoriously unsafe. Well-behaved C++ code is an improvement and NV-Heaps and the `libpmemobj`'s C++ bindings demonstrate that C++ smart pointers and lambdas can provide some of Corundum's checks, but there are several significant gaps. For instance, there is no way to limit the types a lambda can capture.

Go [1] emphasizes simplicity and go-pmem [14] provides basic PMEM programming facilities. However, Go does not allow smart pointers or provide a sufficiently expressive type system to statically enforce the invariants that Corundum provides.

Pony [2] is a new language with similar design goals to Rust. For instance, it has a sophisticated notion of mutability and statically prevents data races, just as Rust does. However, Pony is less mature and more complex than Rust.

3.10 What is Essential and What Is Rust?

The biggest open question for Corundum is what aspects of its design (and our correctness argument) are byproducts of choosing Rust and which represent something more fundamental about persistent memory safety.

For instance, Corundum is unique among PMEM libraries in separating code that modifies persistent state from code that modifies volatile state. Our correctness argument relies on this property in several places, but it is not clear whether this is a fundamentally good idea for PMEM programming or simply something that was helpful in ensuring safety and implementable in Rust.

It would be instructive to try to replicate Corundum's functionality in another language. C++ seems like the most likely mainstream candidate, since it has a flexible type system. Simplying recreating Corundum's approach in C++ would require first recreating the Rust memory safety guarantees that Corundum relies on. This seems challenging. More illuminating would be to try to achieve the same goals using a more idiomatic C++ approach. A comparison between the resulting system and Corundum would be revealing.

4 EVALUATION

We evaluate Corundum along three axes: its success in statically enforcing PM safety properties, its ease of use, and its performance.

4.1 Static Checking

Corundum aims to make the programmer's life easier by statically enforcing PM safety at compile time rather than relying on dynamic checks and testing to identify bugs. To measure its success in this regard, we compare it with other PMEM programming systems.

Table 2 summarizes how Corundum and other PM libraries detect violations of Corundum's six design goals. In the table, "S" (for "Static") means that the compiler either enforces the invariant automatically (e.g., by generating safe code) or detects any violations and reports them, "D" ("Dynamic") means that the system will identify the problem at runtime and exit appropriately, and

App	Rust	Corundum	C++	PMDK
Linked List	192	+19 (9.9%)	146	+45 (30.8%)
Binary tree	256	+12 (4.7%)	208	+41 (19.7%)
HashMap	165	+10 (6.1%)	137	+42 (30.7%)

Table 3: Adding persistence to data structures with Corundum requires fewer changes (measured in lines of code) than PMDK.

"M" ("Manual") means that the system does not detect violations, so they will manifest as a crash, data corruption, or other error. For No-Acyclic-Leaks, "GC" means the system provides garbage collection.

The table shows that Corundum enforces almost all its invariants at compile time, compared to the relatively few compile-time checks other systems provide.

In some cases, this difference represents a design trade-off. For instance, NVM Direct explicitly supports unlogged stores as performance optimization. Likewise, four of the systems allow unsynchronized access which is faster but less safe. A Corundum programmer could use similar techniques to improve performance with `unsafe`.

4.2 Ease of Use

A key goal of Corundum is to make writing safe persistent memory programs easier. Qualitatively, we would expect that the stronger static guarantees that Corundum provides should lead to less debugging. This is especially valuable since many of the bugs that Corundum protects against would manifest during a failure, making them more difficult to test. Our experience using Corundum bears this out: once code compiles, it works reliably. Getting the code to compile can take a while.

Quantitatively, we can measure programing effort by lines of code needed to add persistence to a conventional program. We implemented three data structures in C++ and Rust and then added persistence using PMDK and Corundum. Table 3 shows that Corundum required adding fewer lines in both relative and absolute terms.

4.3 Evaluation Platform

Our test platform has dual 24-core Cascade Lake processors. The CPUs are engineering samples with specs similar to the Xeon Platinum 8160. In total, the system has 384 GB (2 socket × 6 channel × 32 GB/DIMM) of DRAM, and 3 TB (2 socket × 6 channel × 256 GB/DIMM) of Intel Optane DC DIMMs. Our machine runs Fedora 27 with Linux kernel version 4.13.0.

Corundum uses some unstable features of Rust, so we use Rust Nightly version 1.51.0 built with 'release' profile. We compared Corundum with PMDK 1.8, Atlas, Mnemosyne built with '-O2'. All of them use '`clflushopt`' for durability without using non-temporal store. The go compiler applies optimizations to go-pmem by default. We use `Ext4-DAX` to mount the persistent memory and create the pool files.

4.4 Performance

We compared our library with PMDK's libpmemobj and libpmemobj++ by porting some PMDK data structures to Corundum.

BST	A transaction-free and failure-atomic implementation of a Binary Search Tree
KVStore	A simple Key-Value store data structure using hash map
B+Tree	An optimized, balanced B+Tree with 8-way fanout.
wordcount	Counts the occurrences of each word in a corpus of text using a hashmap and producer/consumer threads

Table 4: Microbenchmarks. The first three are used to compare the performance of Corundum with PMDK. Wordcount measures Corundum's scalability with thread count.

Operation	Optane DC		DRAM	
	Mean (ns)	STD (ns)	Mean (ns)	STD (ns)
`Deref`	0.726	-	0.733	-
`DerefMut` (1st)	564.809	699	239.659	123.6
`DerefMut` (not 1st)	0.456	-	0.454	-
`Alloc` (8 B)	529.701	831.6	231.227	130.5
`Alloc` (256 B)	620.378	744.6	249.89	167.4
`Alloc` (4 kB)	1626.167	37715.2	1910.196	2489
`AtomicInit` (8 B)	431.942	371.7	263.67	96.4
`Dealloc` (8 B)	476.851	598.2	223.293	136.5
`Dealloc` (256 B)	581.262	630.8	237.774	145
`Dealloc` (4 kB)	660.553	690.3	242.235	145.7
`TxNop`	258.463	220.2	209.333	59.7
`DataLog` (8 B)	557.517	644.2	249.509	131.1
`DataLog` (2 kB)	607.748	692	267.512	145.6
`DataLog` (32 kB)	2215.264	683.4	1268.671	780.5
`DropLog` (8 B)	31.518	57.8	28.254	15.7
`DropLog` (32 kB)	42.934	111.4	30.754	23
`Pbox::pclone` (8 B)	873.121	652.4	332.809	126.8
`Prc::pclone`	30.132	104.8	24.45	35.9
`Parc::pclone`	56.093	203.3	38.196	58.9
`Prc::downgrade`	21.064	10.5	21.057	6.3
`Parc::downgrade`	33.536	8.2	33.547	6.2
`Prc::Weak:upgrade`	21.801	4.8	21.827	4.6
`Parc::Weak::upgrade`	32.761	0.8	32.826	9.2
`Prc::demote`	49.912	94.8	51.543	113.5
`Parc::demote`	63.235	101.3	64.063	109.5
`Prc::VWeak::promote`	24.235	11	25.21	12.5
`Parc::VWeak::promote`	34.326	32.3	34.122	6.7

Table 5: Corundum's basic operation latency and standard deviation for durability and safety support measured on Intel's Optane DC and Battery-Backed DRAM, with 100K operations per test. I measure a NOP and it returned For all of the , I did it twice: one time measured all 100000 of them + vec::push

4.4.1 Basic Operation Performance. Table 5 reports the latency of basic operations along with the standard deviation measured on the platform described in Section 4.3. To evaluate the impact of storage technology, we measure these operations on both Optane DC persistent memory and DRAM. The standard deviation (STD) shows how much we should expect the measured latency can vary.

To measure dereferencing operation latency, we use a `Pbox<i32>`. Dereferencing a persistent pointer involves address translation and memory indirection. The Rust compiler uses CPU registers to cache the base and offset addresses. As a result, the dereferencing operation performs less than 1 ns, for both read and write. However, writing for the first time requires taking a log of data which takes around 565 ns.

For memory allocation, Corundum uses buddy-blocks algorithm which performs small allocations by splitting large free blocks, and merge small free adjacent blocks on deallocation to yield larger free blocks. Therefore, small free blocks are more available than large free blocks. For example, a free block of size 8192 B can be split into 1024 small free blocks of 8 B, or only 2 large block of size 4 kB. Table 5 confirms this fact. In contrast to allocation, freeing memory takes almost constant time, because the merging happens fewer most of the time due to unavailability of the buddy block. The failure-atomic instantiation operation (AtomicInit) allocates new memory and fill it with a given value atomically using low-level redo logging in the allocator. This operation is as fast as the allocation because the allocation is the only major part of it.

Corundum keeps a journal object in the PM per thread with at list one page of 64 log slots. Therefore, running an empty transaction (TxNop) does not write to the PM. The roughly similar latency in PM and DRAM in Table 5 confirms this. DataLog shows the latency of taking an undo log for a data with 8 B, 2 kB, and 32 kB sizes. It requires allocating memory and copying data to the log location. Therefore, the larger the data, the slower the operation, due to the allocation process. However, creating a DropLog which only keeps the information of allocation has a fixed size, and takes constant time.

The `pclone()` method in `Pbox` creates a new instance of `Pbox` by allocating and copying data to a new location. Therefore, the latency of `Pbox::clone` is the aggregation of PM allocation and `memcpy()`. However, `Prc` and `Parc` do not allocate memory. They only update their reference counters transactionally. `Parc` uses atomic counters which explains its longer latency compared with `Prc`. The `downgrade()` and `upgrade()` function also transactionally update the counters and we can use the same explanation as for `pclone()`. Although the `demote()` function use similar mechanism as `downgrade()` to create volatile weak pointer, they additionally update a reference list in `Prc`/`Parc` which makes them slower. However, the latency of `promote()` is similar to `upgrade()` because they perform the same operation.

4.4.2 Workloads. Table 4 summarizes the workloads we used to evaluate the performance of Corundum and its scalability. The first three applications are used to compare performance with PMDK, Atlas, Mnemosyne, and go-pmem. The PMDK version of BST, KVStore, and B+Tree are available in PMDK repository. We reimplemented them in Corundum and the other libraries using the same algorithms.

4.4.3 Results. Figure 1 shows the results of our experiments comparing Corundum's performance with PMDK, Atlas, Mnemosyne, and go-pmem. Performance with pool types ("Corundum") is always at least as fast as other libraries, and sometimes significantly faster.

Wordcount measures scalability. Corundum provides a separate allocator and journal object for every thread to allow concurrency. Figure 2 confirms that the performance scales with thread count. When the allocator cannot find a free block in its free lists, it deliberates the task to another allocator which is protected by a lock.

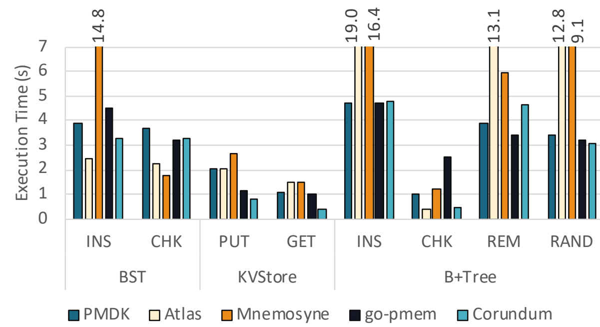

Figure 1: Performance comparison between Corundum, PMDK, Atlas, Mnemosyne, and go-pmem

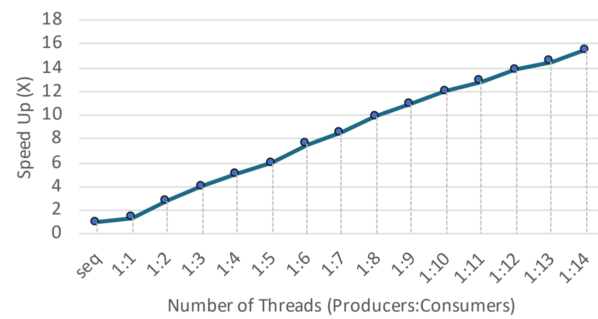

Figure 2: Corundum's scalable performance with regard to the number of threads. The baseline is running one producer and one consumer object sequentially (seq).

5 RELATED WORK

Many projects have addressed the challenges of PM programming with software [4, 8, 9, 11, 12, 16, 18, 21, 22, 25, 26, 33, 34, 36] and/or hardware [17, 29, 37, 39]. Table 2 highlights the checks some of these systems provide. The systems not listed in table provide fewer checks (in some cases this is by design).

In response, several projects provide testing [23, 24, 31] and debugging tools [3, 20, 30] targeting PM systems. While useful, these tools cannot provide safety guarantees and they rely on the programmer using them reliably and providing tests with good coverage.

5.1 PM Programing Libraries

Among PM libraries, PMDK [33] is the mostly widely used. It provide dynamic checks to prevent inter-pool pointers in C and the C++ version provides static enforcement for atomicity, but otherwise the programmer is responsible for enforcing safety.

NV-Heaps [11] and Mnemosyne [36] rely on the C++ type system and a custom compiler, respectively, to statically avoid data races, atomicity violations, and pointers from PM into volatile memory. Both systems go to some pains help avoid bugs, but the weakness of the C/C++ type systems make the guarantees they provide easy to circumvent. NV-heaps addresses the problem of closed pools – by not allowing pools to close.

NVM Direct [8] adds extensions to C/C++ via a custom compiler that gives the programmer detailed control over logging. It is, by design, "dangerously flexible - just like C" [7] to enable as many manual optimizations as possible, so it relies heavily on the programmer to enforce safety properties. Corundum opts for safety over flexibility and does not require specific compiler support.

5.2 Orthogonal Persistence

The fundamentals of persistent programming have been studied for many years, and the notion of orthogonal persistence has been proposed as a guiding principle in PMEM software design [5]. The principle holds that the persistence abstraction should allow the creation and manipulation of data in an identical manner , regardless of its (non)persistence – that is that persistence should be *orthogonal* to other aspects of the language.

Despite the attractivensess of orthogonal persistence, Corundum and other recently-proposed system are not orthogonally persistent. We made this design decision in Corundum because all three principles of orthogonal persistence [5] face practical problems, especially with respect to performance, system complexity, and consistency. First, "persistence independence" (i.e., using the same code for transient and persistent data) is slow and/or complicated, especially in low-level language like Rust. Persistent operations require different (and slower) instructions than operations on transient data, and an orthogonal system would require a runtime mechanism to choose which version to run or would need to always run the slow persistent code. Either choice will hurt performance. Second, "data type orthogonality" (i.e., any data type can be persistent) leads to consistency problems since some types (e.g., network sockets or file handles) are inherently transient. The final principle of "persistence identification" (i.e., not expressing persistence in the type system) leads to complications in systems with multiple pools of persistent memory, since the question is not "transient or persistent" but "transient and, if not, in which pool". Without type information, it seems very challenging to statically prevent the creation of inter-pool pointers as Corundum does.

Furthermore, when orthogonal persistence was first formulated, persistence required a disk, so the performance cost of orthogonality was not an issue. For persistent memory, those costs would be prohibitive.

6 CONCLUSION

Corundum enforces PM safety invariants using mostly static checks. It, therefore, eliminates memory management, pointers safety, and logging bugs and avoids the attendant costs of testing, debugging, fixing, and recovering from them. It accomplishes this using Rust's type system to carefully control when the program can modify persistent and volatile state and when and where mutable references to persistent state can exist. Our experience shows that Corundum relatively easy to use and our measurements show that Corundum's performance is as good as or better than PMDK.

REFERENCES

[1] [n.d.]. *The Go Programming Language.*
[2] [n.d.]. *Pony programming language.*
[3] 2015. *An introduction to pmemcheck.*
[4] Mohammad Alshboul, James Tuck, and Yan Solihin. 2018. Lazy persistency: A high-performing and write-efficient software persistency technique. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* IEEE, 439–451.
[5] Malcolm Atkinson and Ronald Morrison. 1995. Orthogonally persistent object systems. *The VLDB Journal* 4, 3 (1995), 319–401.
[6] Emery D Berger, Ting Yang, Tongping Liu, and Gene Novark. 2009. Grace: Safe multithreaded programming for C/C++. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications.* 81–96.
[7] Bill Bridge. [n.d.]. personal communication.
[8] Bill Bridge. 2015. NVM Support for C Applications. Available at http://www.snia.org/sites/default/files/BillBridgeNVMSummit2015Slides.pdf.
[9] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14).* ACM, New York, NY, USA, 433–452. https://doi.org/10.1145/2660193.2660224
[10] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* 8, 5 (2015), 497–508.
[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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* (Newport Beach, California, USA) *(ASPLOS '11).* ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380
[12] Nachshon Cohen, David T Aksun, and James R Larus. 2018. Object-oriented recovery for non-volatile memory. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–22.
[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14).* ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814
[14] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. 2020. go-pmem: Native Support for Programming Persistent Memory in Go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20).* 859–872.
[15] Vincent Gramoli, Rachid Guerraoui, and Vasileios Trigonakis. 2012. TM2C: a software transactional memory for many-cores. In *Proceedings of the 7th ACM european conference on Computer Systems.* 351–364.
[16] Jinyu Gu, Qianqian Yu, Xiayang Wang, Zhaoguo Wang, Binyu Zang, Haibing Guan, and Haibo Chen. 2019. Pisces: A Scalable and Efficient Persistent Transactional Memory. In *2019 USENIX Annual Technical Conference (USENIX ATC 19).* USENIX Association, Renton, WA, 913–928. https://www.usenix.org/conference/atc19/presentation/gu
[17] Jungi Jeong, Chang Hyun Park, Jaehyuk Huh, and Seungryoul Maeng. 2018. Efficient hardware-assisted logging with asynchronous and direct-update for persistent memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 520–532.
[18] J. Hyun Kim, Young Je Moon, Hyunsub Song, Jay H. Park, and Sam H. Noh. 2020. On Providing OS Support to Allow Transparent Use of Traditional Programming Models for Persistent Memory. *J. Emerg. Technol. Comput. Syst.* 16, 3, Article 33 (June 2020), 24 pages. https://doi.org/10.1145/3388637
[19] Steve Klabnik and Carol Nichols. [n.d.]. *The Rust Programming Language.*
[20] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *USENIX Annual Technical Conference (ATC).*
[21] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. 2018. iDO: Compiler-directed failure atomicity for nonvolatile memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* IEEE, 258–270.
[22] Sihang Liu, Korakit Seemakhupt, Gennady Pekhimenko, Aasheesh Kolli, and Samira Khan. 2019. Janus: Optimizing Memory and Storage Support for Non-Volatile Memory Systems. In *Proceedings of the 46th International Symposium on Computer Architecture* (Phoenix, Arizona) *(ISCA '19).* Association for Computing Machinery, New York, NY, USA, 143–156. https://doi.org/10.1145/3307650.3322206
[23] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems.* 1187–1202.
[24] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A fast and flexible testing framework for persistent memory programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.* 411–425.
[25] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. 2018. PM: Simplifying Application Usage of Persistent Memory. *ACM Trans. Storage* 14, 4,

Article 34 (Dec. 2018), 18 pages. https://doi.org/10.1145/3278141

[26] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. 2020. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 789–806.

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

[28] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.* 17, 1 (1992), 94–162. https://doi.org/10.1145/128765.128770

[29] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. 2018. Steal but no force: Efficient hardware undo+ redo logging for persistent memory systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 336–349.

[30] Kevin Oleary. 2018. *How to detect persistent memory programming errors using Intel Inspector - Persistence Inspector.*

[31] Ismail Oukid, Daniel Booss, Adrien Lespinasse, and Wolfgang Lehner. 2016. On testing persistent-memory-based software. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*. 1–7.

[32] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2014. Software engineering with transactional memory versus locks in practice. *Theory of Computing Systems* 55, 3 (2014), 555–590.

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

[34] Thomas Shull, Jian Huang, and Josep Torrellas. 2019. AutoPersist: An Easy-to-Use Java NVM Framework Based on Reachability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 316–332. https://doi.org/10.1145/3314221.3314608

[35] stack overflow [n.d.]. https://insights.stackoverflow.com/survey/2019#most-loved-dreaded-and-wanted

[36] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA). ACM, New York, NY, USA.

[37] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. 2020. Hardware-Based Domain Virtualization for Intra-Process Isolation of Persistent Memory Objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 680–692.

[38] Zhen Yu, Yu Zuo, and Yong Zhao. 2020. Convoider: A Concurrency Bug Avoider Based on Transparent Software Transactional Memory. *International Journal of Parallel Programming* 48, 1 (2020), 32–60.

[39] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture* (Davis, California) *(MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744

A ARTIFACT APPENDIX

A.1 Abstract

This appendix provides the necessary information for obtaining the source code, building, and running performance and functionality tests of Corundum. We describe the hardware and software requirements to run the experiments and reproduce the results as they appear in the Section 4.

A.2 Artifact check-list (meta-information)

- **Program:** Corundum library and its unit tests, Rust, Cargo, PMDK v1.8. Input files for the experiments are included (74 MB).
- **Compilation:** Rust 1.50.0-nightly (publicly available), GNU C++11, CMake >= 3.3
- **Data set:** Data set is included (`eval/inputs`)
- **Run-time environment:** Linux (latest version). The NVMM should be mounted in `/mnt/pmem` using a DAX-enabled file system (e.g. EXT4-DAX). We also provide a docker image containing linux 20.04 LTE with pre-installed dependencies.
- **Hardware:** A machine with a 16-core Intel processor with 16 GB or larger NVMM (e.g. Optane DC). The docker does not require NVMM and uses DRAM to emulate NVMM.
- **Metrics:** Execution time
- **Output:** Numerical results stored in `perf.csv` and `scale.csv` for performance and scalability, respectively. Expected results are also included in the text.
- **Experiments:** A script is provided to run the experiments and generate results. Although the results may vary depending on the build and the environment, there should not be a big difference between Corundum's performance and PMDK's.
- **How much disk space required (approximately)?:** 32 GB
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes
- **How much time is needed to complete experiments (approximately)?:** $\leq$ 1 hour.
- **Publicly available?:** Code, unit-tests, documentation, examples, and evaluation scripts are publicly available.
- **Code licenses (if publicly available)?:** Apache v2.0
- **Archived (provide DOI)?:** DOI: 10.5281/zenodo.4329841

A.3 Description

A.3.1 How to access. The artifacts are publicly available through Zenodo archival repository and GitHub. You can access the code by using its DOI or cloning the GitHub repository at https://github.com/NVSL/Corundum/.

We also prepared a Docker image available

A.3.2 Hardware dependencies. We recommend running the experiments on a machine with physical persistent memory such as Intel Obtane DC with at least 16 GB available space.

Also, the scalability tests require at least 16-core processor to measure the execution time while running threads in parallel.

A.3.3 Software dependencies.

- If you wish to use Docker, please install Docker and use the prepared docker image with pre-installed dependencies. Otherwise, please run `eval/setup.sh` on Ubuntu latest version.
- To measure performance, we use `perf`, a linux builtin monitoring tool for analyzing programs (included in `eval/setup.sh`).
- To compare Corundum with PMKD, please install both 'libpmemobj' and 'libpmemobj-cpp' by running `eval/build.sh`.

A.4 Installation

The docker image already have the dependencies pre-installed. If you wish to run it on a real system, please follow the steps in the rest of this section.

A.4.1 Installing Rust. If you prefer installing Rust manually rather than running '`setup.sh`', please follow throw these steps. On a Unix-like OS machine, the following commands install Rust compiler, Cargo, and `rustup`.

```
curl --proto '=https' --tlsv1.2 \
     -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
rustup default nightly
```

If tou are using another OS, please refer to Rust's official website for the instructions at

`https://www.rust-lang.org/tools/install`

A.4.2 Building Corundum. Corundum is publicly available in GitHub. Please use the following commands to clone and compile it.

```
$ git clone https://github.com/NVSL/Corundum.git
$ cd Corundum
$ cargo build --release --examples
```

A.4.3 Verifying the compilation. Optionally you can run the tests to verify it. Since tests may share pool files, we run them sequentially by using a single thread.

```
cargo test --tests -- --test-threads=1
```

A.5 Experiment workflow

Running the experiments is automated though the following set of commands (for your convenience, please login as 'root').

```
$ git clone https://github.com/NVSL/Corundum.git
$ cd Corundum/eval
$ ./setup.sh     # Install dependencies
$ ./build.sh     # Build the workloads
$ ./run.sh       # Run the tests
$ ./results.sh   # Display the results
```

The `eval/setup.sh` downloads necessary libraries (for PMDK and Rust), and `eval/build.sh` compiles the benchmark applications. `eval/run.sh` executes benchmarks and collects the results, and generates two files:

- `perf.csv`: Compares Corundum with PMDK (Figure 1)
- `scale.csv`: Evaluates multi-threading scalability (Figure 2)

Finally, `eval/results.sh` displays the results on screen.

A.6 Running on a Docker

We also prepared a docker image with pre-installed dependencies, PMDK (libpmemobj and libpmemobj-cpp), Corundum, and the input datasets. We will use `tmpfs` in the docker to mount DRAM into '`/mnt/pmem0`'. We also need to enable `perf_event_open` system calls. Please run the following commands on the host machine to give permission to, download, and run the docker image.

```
$ sudo sysctl -w kernel.perf_event_paranoid=-1
$ wget https://raw.githubusercontent.com/NVSL/\
Corundum/main/eval/docker-default.json
$ docker run --security-opt \
    seccomp=./docker-default.json \
    --mount type=tmpfs,destination=/mnt/pmem0 \
    -it mhz88/corundum:latest bin/bash
```

Inside the docker, please run the following commands to run the experiments.

```
$ cd ~/Corundum/eval
$ ./run.sh && ./results.sh
```

Execution Time (s)								
	BST		KVStore		B+Tree			
	INS	CHK	PUT	GET	INS	CHK	REM	RAND
PMDK	4.5	4.1	8.7	5.1	8.6	2.2	7.9	6.3
Corundum	3.4	3.4	2.9	2.1	12.9	1.4	11.8	8.6

Table 6: Expected output for performance comparison between PMDK and Corundum (perf.csv).

A.7 Evaluation and expected results

We discuss performance evaluation in terms of execution time compared with an equivalent implementation in PMDK, and execution time using multiple threads to show scalability.

A.7.1 Performance. To compare Corundum's performance with PMDK, we use three applications written in both of them: BST, VKStore, and B+Tree. The provided script runs these applications automatically with random inputs. The results are stored in `perf.csv` and should look like Table 6.

A.7.2 Scalability. To verify that Corundum provides scalability with respect to the thread number, we implemented a MapReduce application called `grep` which counts the frequency of every word in a list of text documents. The producer threads fill up a shared stack, and the consumer threads pop a batch from the stack, using a local record, count the frequency of every words in the batch, and finally update a shared HashMap with the local records. Table 7 shows the expected execution times in seconds for various number of producer (p) and consumer (c) threads. We use Large Canterbury Corpus (http://www.data-compression.info/Corpora/CanterburyCorpus/) dataset as the input files (included in the archive).

p/c	1	2	3	7	14
1	30.83	15.85	11.09	7.45	8.03
2	28.77	15.65	11.65	6.96	7.36

Table 7: Expected output for Scalability (scale.csv) using emulated PM.

A.8 Experiment customization

Corundum requires a DAX-enabled file system. We recommend EXT4-DAX. If you run the experiments on a real machine, use the following commands to format and mount the derive.

```
mkfs.ext4 /dev/pmem0
mount -o dax /dev/pmem0 /mnt/pmem0
```

Many of the latency components are originated in Hardware, such as using `CL_FLUSHOPT` instead of `CL_FLUSH` when available. Corundum does not automatically detect these capabilities. However, it comes with some builtin features. In '`Cargo.toml`', under the 'features' section, update `default=[]` as required. For example, if the system supports `CLWB` instructions, you may force Corundum to use that by changing the default features to this:

```
default = ["pthread", "use_clwb"]
```

Please note that in `run.sh`, we disabled `CL_FLUSHOPT` and `CLWB` to force both PMDK and Corundum use `CL_FLUSH` to persist data.

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html