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 all 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 as or better than Intel’s widely-used PMDK library, HP’s Atlas, Mnemosyne, and go-pmem.

CCS CONCEPTS
- Information systems → Storage class memory; - Software and its engineering → Formal software verification; Software testing and debugging; - Hardware → Non-volatile memory.

KEYWORDS
non-volatile memory programming library, static bug detection, crash-consistent programming

ACM Reference Format:

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 comparable 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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASPLOS ’21, April 19–23, 2021, Virtual, USA  
© 2021 Copyright held by the owner/author(s).  
https://doi.org/10.1145/3445814.3446710
create new types of smart pointers that integrate cleanly with the rest of the language. Its type system 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 static checking (in almost all cases) to enforce 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, each 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 [7, 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 object from which other objects

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 variables’ 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, Immutability, 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 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<Box<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` 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 Mpenyosync [36]. Corundum provides four abstractions – typed persistent memory pools, transactions, persistent smart pointers,
and atomic memory allocation—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 in acyclic structures 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 [7, 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::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`.

Reference and raw pointers, and types that refer to state external to the program (e.g., file handles) are `!PSafe`. Having a `!PSafe` field also makes a type `!PSafe`. As a result, `Box`, `Rc`, `Arc`, `Mutex`, and the other types in Rust containing raw pointers 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 is a reference to a `journal object` that holds information about the current transaction. Variable `j` is of type `Journal<PS>`, 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.

**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,
1 #[derive(Root)]
2 struct Node {
3   val: i32,
4   next: PRefCell<Pbox<Node,P>>, P>
5 }
6 fn append(n: &Node, v: i32, j: &Journal<P>) {
7   let mut t = n.next.borrow_mut(j);
8   match &t {
9     Some(succ) => append(succ, v, j);
10     None => *t = Some(Pbox::new(
11       Node {
12         val: v,
13         next: PRefCell::new(None, j)
14       }, j));
15   }
16 }
17 fn go(v: i32) {
18   let head = P::open::<Node>("list.pool", 0);
19   P::transaction(|| {
20     append(&head, v, j);
21   });
22 }
Listing 1: A Corundum implementation of linked list append. Some error management code has been elided for clarity.

let mut done = false;
let p1 = P::transaction(|| {
  let p1 = Pbox::new(10);
  let p2 = Pbox::new(2, j);
  root.set(p2);
  done = true;
  ^^^ the trait `TxInSafe` is not implemented for `&mut bool`
  p1
  ^ the trait bound `Pbox<i32,P>::TxOutSafe` is not satisfied
}).unwrap();

Listing 2: Reachability though lifetime and type bounding

Variables can be created and modified within the transaction. Pre-existing volatile data can be read.

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: Statically-Enforced Persistent Memory Safety

1 P1::transaction(|| {
2   let v = Box::new(10);
3   let p1 =
4     Pbox::new(v, j);
5   ^ Box<i32>::PSafe is not satisfied
6 }).unwrap();

Listing 3: Only persistent-safe objects are acceptable.

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 (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 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.

`VWeak` is the only way to hold a volatile pointer to persistent data. 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
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.

<table>
<thead>
<tr>
<th>Corundum type</th>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PMEM Smart Pointers for Dynamic Allocation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>plclone(j: &amp;Journal&lt;P&gt;)</td>
<td>Allocate PMEM in P and initialize it.</td>
</tr>
<tr>
<td></td>
<td>downgrade() -&gt; PWeak</td>
<td>Create a new reference to the data.</td>
</tr>
<tr>
<td></td>
<td>demote() -&gt; VWeak</td>
<td>Return a persistent weak (PWeak&lt;T,P&gt;).</td>
</tr>
<tr>
<td>PWek&lt;T,P&gt;</td>
<td>upgrade() -&gt; Prc/Parc</td>
<td>Convert to a Option&lt;Prc/Parc&lt;T,P&gt;&gt; if it is available.</td>
</tr>
<tr>
<td>VWeak&lt;T,P&gt;</td>
<td>promote() -&gt; Prc/Parc</td>
<td>Convert to a Option&lt;Prc/Parc&lt;T,P&gt;&gt; if it is available.</td>
</tr>
</tbody>
</table>

| **PMEM Wrappers for Interior Mutability**                                                                                                 |
| PCell<T,P>          | new(value: T, j: &Journal<P>) | Interior mutability via copying data to and from PMEM using get() and set() functions.           |
|                     | borrow()          | Interior mutability via references with dynamic borrow checking.                                  |
|                     | borrow_mut(j: &Journal<P>) | Create new instance on the stack and initialize it.                                               |
|                     | lock(j: &Journal<P>) | Return an immutable reference to value it contains.                                               |
|                     | upgrade() -> Prc/Parc | Return a mutable reference object (RefMut) for the value inside.                                 |
|                     | lock(j: &Journal<P>) | Create new instance on the stack and initialize it.                                               |

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 VWeak is the only way store a pointer to PMEM in volatile memory. The object a VWeak refers to can disappear if the last Prc/Parc referring 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, and a pool will not close while a transaction is in progress.

Second, consider a reference (other than a VWeak), A, that exists outside a transaction. Rust’s borrow checker requires that a chain of in-scope references exists from the root of the pool 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 released 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.
Listing 4: Cross-Pool referencing prevention via type system

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 PMutex 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-Non-Volatile-Mutability).

Design Goal 4 (No-Races) Holds: Rust prevents data races and unsynchronized access using the mutability invariant, the Mutex type, and 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-Non-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.

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 Pbox provide a method, pclone(), to create a new reference to the shared data and increment the reference count. The reference counts are persistent and must be logged, so pclone() takes a journal object argument.

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 Prc, 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 to prevent persistent memory leaks (in the absence of cycles) and multiple-frees. Corundum provides an additional guarantee that is necessary for persistent memory: It ensures that, by the end of the allocating transaction, the newly allocated persistent memory is reachable from some previously allocated persistent memory. This ensures that crashes (which invalidate all volatile pointers) do not leak volatile memory.

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

Corundum adds three mechanisms extending Rust’s reference counting mechanism to prevent memory leaks in acyclic persistent data structures. 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.

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:

```rust
let a = Parc::new(j, 42);
thread::spawn(move |k| {
    let b = a;
});
```

Rust’s `thread::spawn()` function executes its argument (a lambda) in a new thread. In the code, `a` is an orphan but it moves to the thread’s scope. The thread’s body is outside the transaction, so our invariant does not hold. To prevent this, Corundum makes `PBox` and `Parc` `Send` to prevent their capture by or transmission to another thread. `VWeak` remains `Send`, so it can be used to transmit persistent state between threads.

### 3.8 Example

Listing 1 implements `append()` for a persistent linked list in Corundum. A `Node` contains a link to the next `Node`. The link is of type `Parc::VWeak<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. Lines 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 `Option`’s content to `Node` to `succ`, and recursively calls `append()`.

If the `Option` is `None`, the code has reached the end of the `Node` list. Line 10 creates a `PBox` to allocate a new `Node` with value `v` and a next `Node` 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 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 the algorithm in our example correctly appends to a linked list).

**Dynamic Checks** In some cases, Corundum provides dynamic, rather than static, checks. In these cases 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 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 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 enforce crash consistency. More permissive libraries allow this, but such code would not compile under 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.
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)

<table>
<thead>
<tr>
<th>System</th>
<th>Obj-F-Object</th>
<th>Pts-Are-Safe</th>
<th>Pts-Are-Safe</th>
<th>No-Races</th>
<th>TX-Are-Atomic</th>
<th>No-Leaks</th>
</tr>
</thead>
<tbody>
<tr>
<td>libpmemobj [33]</td>
<td>M</td>
<td>D</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>libpmemobj++ [33]</td>
<td>M</td>
<td>D</td>
<td>M</td>
<td>M</td>
<td>M</td>
<td>M</td>
</tr>
<tr>
<td>Corundum</td>
<td>S</td>
<td>S/D</td>
<td>S</td>
<td>D</td>
<td>S</td>
<td>S</td>
</tr>
</tbody>
</table>

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 was 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. 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.

4 EVALUATION

We evaluate Corundum along three axes: its success in statically enforcing PM safety properties, its ease of use, and its performance.
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 1.52.0 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 Atlas, Mnemosyne, go-pmem, and Corundum.

#### 4.4.1 Basic Operation Performance

Table 5 reports the latency of basic operations 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.

To measure dereferencing operations, we use a Pbox<i32>. Dereferencing a persistent pointer involves address translation and memory indirection. The Rust compiler tends to keep the base and offsets for address translation in registers. As a result, the dereferencing operation performs less than 1 ns, for both read and write. However, writing for the first time requires logging which takes around 470 ns (DerefMut, the 1st time).

For memory allocation, Corundum uses a buddy system [20] which performs small allocations by splitting large free blocks, and coalescing 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 merging is rare.

The failure-atomic instantiation operation (AtomicInit) allocates new memory and fills 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 preallocates a per-thread journal object, so running an empty transaction (TxNop) does not write to the PM.

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

---

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>PMDK</th>
<th>Corundum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc (8 B)</td>
<td>375</td>
<td>241</td>
</tr>
<tr>
<td>Alloc (256 B)</td>
<td>706</td>
<td>264</td>
</tr>
<tr>
<td>Alloc (4 kB)</td>
<td>1685</td>
<td>1907</td>
</tr>
<tr>
<td>DerefMut (the 1st time)</td>
<td>467</td>
<td>235</td>
</tr>
<tr>
<td>DerefMut (not the 1st time)</td>
<td>0.4</td>
<td>0.4</td>
</tr>
</tbody>
</table>

**Table 5: Corundum’s basic operation latency for durability and safety support measured on Intel’s Optane DC and Battery-Backed DRAM, with 50K operations per test.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Optane DC Avg (ns)</th>
<th>DRAM Avg (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deref</td>
<td>0.9</td>
<td>1.0</td>
</tr>
<tr>
<td>DerefMut (the 1st time)</td>
<td>467</td>
<td>235</td>
</tr>
<tr>
<td>DerefMut (not the 1st time)</td>
<td>0.4</td>
<td>0.4</td>
</tr>
<tr>
<td>Alloc (8 B)</td>
<td>734</td>
<td>241</td>
</tr>
<tr>
<td>Alloc (256 B)</td>
<td>706</td>
<td>264</td>
</tr>
<tr>
<td>Alloc (4 kB)</td>
<td>1685</td>
<td>1907</td>
</tr>
<tr>
<td>Dealloc (8 B)</td>
<td>632</td>
<td>384</td>
</tr>
<tr>
<td>Dealloc (256 B)</td>
<td>598</td>
<td>249</td>
</tr>
<tr>
<td>Dealloc (4 kB)</td>
<td>675</td>
<td>248</td>
</tr>
<tr>
<td>Pbox:AtomicInit (8 B)</td>
<td>822</td>
<td>416</td>
</tr>
<tr>
<td>Prc:AtomicInit (8 B)</td>
<td>2089</td>
<td>1210</td>
</tr>
<tr>
<td>ParcAtomicInit (8 B)</td>
<td>3508</td>
<td>2304</td>
</tr>
<tr>
<td>TxNop</td>
<td>198</td>
<td>198</td>
</tr>
<tr>
<td>DataLog (8 B)</td>
<td>574</td>
<td>253</td>
</tr>
<tr>
<td>DataLog (1 kB)</td>
<td>1070</td>
<td>500</td>
</tr>
<tr>
<td>DataLog (4 kB)</td>
<td>2463</td>
<td>1510</td>
</tr>
<tr>
<td>DropLog (8 B)</td>
<td>29</td>
<td>28</td>
</tr>
<tr>
<td>DropLog (32 kB)</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>Pbox::pclone (8 B)</td>
<td>698</td>
<td>314</td>
</tr>
<tr>
<td>Prc::pclone</td>
<td>13</td>
<td>8</td>
</tr>
<tr>
<td>Parc::pclone</td>
<td>333</td>
<td>170</td>
</tr>
<tr>
<td>Prc::downgrade</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Parc::downgrade</td>
<td>335</td>
<td>174</td>
</tr>
<tr>
<td>Prc::PWeak::upgrade</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Parc::PWeak::upgrade</td>
<td>321</td>
<td>175</td>
</tr>
<tr>
<td>Prc::demote</td>
<td>42</td>
<td>43</td>
</tr>
<tr>
<td>Parc::demote</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Prc::VWeak::promote</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>Parc::VWeak::promote</td>
<td>352</td>
<td>175</td>
</tr>
</tbody>
</table>
DropLog which only keeps the information of the allocation, and takes constant time.

Pbox::pclone() creates a new instance of Pbox by allocating and copying data to a new location. Therefore, the latency of Pbox::pclone combines PM allocation and memcpy(). However, Prc and Parc do not allocate memory. They only update their reference counters transactionally. Parc takes a log every time it increments or decrements to provide crash-consistent atomic counters which explains its longer latency compared with Prc. The downgrade() and upgrade() functions 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 slightly 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. Corundum’s performance is almost as fast as other libraries, and sometimes significantly faster.

The Wordcount benchmark measures scalability. It uses a single IO thread to pull text from input files and distributes it to a pool of worker threads that count words. It is embarrassingly parallel. Corundum provides a separate allocator and journal object for every thread to allow concurrency. Figure 2 shows that the performance scales with thread count, demonstrating that Corundum does not limit scalability.

5 RELATED WORK

Many projects have addressed the challenges of PM programming with software [4, 7, 9, 11, 12, 16, 18, 22, 23, 26, 33, 34, 36, 37] 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 [24, 25, 31] and debugging tools [3, 21, 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 Programming Libraries

Among PM libraries, PMDK [33] is the most widely used. It provides 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 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 [7] 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” [8] 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 attractiveness of orthogonal persistence, Corundum and other recently-proposed systems are not orthogonally persistent. We made this design decision in Corundum because all
three principles of orthogonal persistence 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 mostly using 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 is relatively easy to use and our measurements show that Corundum’s performance is comparable with (or better than) existing PM libraries.

ACKNOWLEDGEMENT

This work was supported in part by Semiconductor Research Corporation (SRC) and by NSF award 1629395.

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 Section 4.

A.2 Artifact Check-List (Meta-Information)

- **Program**: Corundum library and its unit tests, Rust, Cargo, PMDK v1.8, Atlas, Mnemosyne, and go-mem. Input files for the experiments are included (74 MB).
- **Compilation**: Corundum: Rust 1.52.0-nightly (publicly available); PMDK, Atlas, Mnemosyne: GNU C++11, CMake >= 3.3; go-pmem: Go lang.
- **Data set**: Data set is included (Will be downloaded automatically)

- **Run-time environment**: Linux (tested in Ubuntu 20.04, Fedora 27, and NixOS). The NVMM should be mounted in /mnt/pmem0 using a DAX-enabled file system (e.g. EXT4-DAX). We also provide a docker image containing Ubuntu 20.04 LTE with pre-installed dependencies.
- **Hardware**: A machine with a 16-core Intel processor with 32 GB or larger NVMM (e.g. Optane DC). The docker does not require NVMM and uses DRAM to emulate NVMM. The ISA should provide CL_FLUSHOPT since we customized Atlas to use it instead of CL_FLUSH. Other libraries are adaptable to the system configuration.
- **Metrics**: Execution time
- **Output**: Numerical results stored in perf.csv, scale.csv, and micro.csv for performance, scalability, and micro-benchmarks, 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.
- **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)**: ≤ 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.4539743

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 at DockerHub (mhz88/corundum:latest).

A.3.2 Hardware Dependencies. We recommend running the experiments on a machine with a physical persistent memory such as Intel Optane DC with at least 32 GB available space.

Also, the scalability test requires a 16-core processor to measure the execution time while running threads in parallel.

A.3.3 Software Dependencies.

- To install Corundum without running the evaluations, installing Rust compiler with Cargo tool (1.52.0-nightly) would suffice.
- To measure performance, we use perf (>=4.9.215), a Linux (kernel>=3.10.0) built-in monitoring tool for analyzing programs (automated installation is included in ‘eval/ubuntu-deps.sh’).
- If you wish to use Docker, please install Docker and use the prepared docker image with pre-installed dependencies. Otherwise, please run os specific setup scripts (e.g. ‘eval/ubuntu-deps.sh’ on Ubuntu latest version).
- ‘eval/build.sh’ compiles and installs other libraries and workloads for comparison. It internally calls ‘build.sh’ scripts for each individual library (can be found in ‘eval/pmdk’, ‘eval/atlas’, ‘eval/mnemosyne’, and ‘eval/go’).

A.4 Installation

The docker image already has 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 ‘ubuntu-deps.sh’, please follow through these steps. On a Unix-like OS machine, the following commands install Rust compiler, Cargo, and rustup.
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:

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

A.5 Experiment Workflow

Running the experiments is automated through the following set of commands:

```
$ sudo sysctl -w kernel.perf_event_paranoid=-1
$ git clone https://github.com/NVSL/Corundum.git
$ cd Corundum/eval
$ ./ubuntu-deps.sh # Install dependencies
$ ./build.sh # Build the libraries and workloads
$ ./run.sh -o # Run the tests with CLFlushOPT
$ ./results.sh # Display the results
```

The eval/ubuntu-deps.sh downloads necessary dependencies, and eval/build.sh compiles the benchmark applications. eval/run.sh executes benchmarks and collects the results, and generates three files:

- perf.csv: Compares Corundum with other libraries (Figure 1)
- scale.csv: Evaluates multi-threading scalability (Figure 2)
- micro.csv: Lists average latency for basic operations (Table 5)

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

A.6 Evaluation on Docker

We also prepared a docker image with pre-installed dependencies, PMDK with equivalent implementations in PMDK, Atlas, Mnemosyne, go-pmem, Corundum, and the input datasets. To emulate PM, we bind /dev/shm to a directory inside the docker (i.e., `-v /dev/shm:/mnt/pmem0`). To use real PM, you may mount it to a folder as explained in Section A.8, and then replace `/dev/shm` in the command line arguments with the mount point directory (e.g., `-v /mnt/pmem0:/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
$ git clone https://raw.githubusercontent.com/NVSL/Corpora/CanterburyCorpus/) dataset as the input files
```

A.7 Evaluation and Expected Results

We discuss performance evaluation in terms of execution time compared with equivalent implementations in PMDK, Atlas, Mnemosyne, go-pmem, and execution time using multiple threads to show its scalability.

A.7.1 Performance. To compare Corundum’s performance with other libraries, we use three applications: 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 data in Figure 1. Please modify other library installations as needed since we do not guarantee them.

A.7.2 Scalability. To verify that Corundum provides scalability with respect to the number of threads, we implemented a MapReduce application called grep (referred as wordcount in the text) 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 segment from the stack and count the number of appearance of every words in the segment locally. We do not collect the local records because it adds a fixed amount of execution time to sequentially fetch data from threads. This is to better evaluate the scalability of the library. Figure 2 shows the expected speedup in execution time 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).

A.7.3 Microbenchmarks. The run script also reproduces Table 5. To measure the latency of each individual operation, we use a Rust provided measurement tool called Instant and Duration. On Linux, it internally uses clock_gettime() system call. Since this tool is not zero cost, for operations with low latency such as Deref, we measure running a batch of 50k of them and then calculate the average number. For these operations, we cannot report standard deviation.

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 drive (assuming the device name is `/dev/pmem0`):

```
$ sudo mkfs.ext4 /dev/pmem0
$ sudo mknod -t ext4 /dev/pmem0 /mnt/pmem0
```

A high-performing and write-efficient software persistency technique. In 2018

![In this case, the figure or table should be included here.]

REFERENCES


basic.html.