Rust for Persistent Memory programming

Fine-grained access in a transactional context

Louis Boulanger, Frédéric Wagner, Yves Denneulin

Université Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG

25th March 2022

The objective of this presentation

Safe, programmer-friendly NVRAM API in Rust?

Introduction
Persistent memory
Rust

Rust and NVRAM: Taenite

The MCell

Experiments

Conclusion

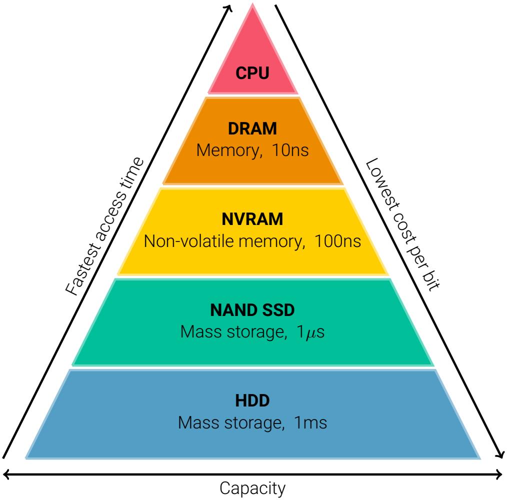
Persistent memory: the basics

Persistent memory (PMEM, NVM, NVRAM):

- Byte-addressable (RAM)
- Persistent after power loss (SSD, HDD)

→ Unique constraints

Persistent memory: cost



Persistent memory: use cases

NVRAM can be used as:

- A RAM extension (persistence is ignored)
- A cache for SSDs (byte-addressability is ignored)
- Persistent storage for un-serialized application data

Persistent memory: ensuring persistency

Several x86 instructions exist to ensure memory write persistency:

- Flushes: `CLFLUSH[OPT]`, `CLWB`
  - Flushes a *cache line*, applying modifications to actual NVRAM.
- Fences: `SFENCE`, `LFENCE`, `MFENCE`
  - Order memory **S**tores, **L**oads or both (**M**): everything before must be done before everything after

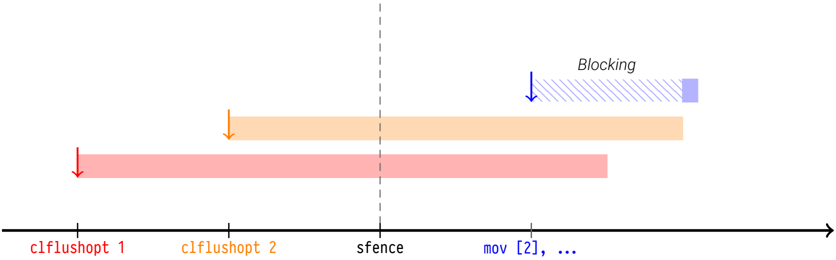
Persistent memory: ensuring persistency

A small example in x86 assembly:

```
mov         [nvram_addr_1], 1
clflushopt  nvram_addr_1

mov         [nvram_addr_2], 2
clflushopt  nvram_addr_2

sfence

mov         [nvram_addr_2], ...
```

The order of the writes before the `SFENCE` isn't defined: the second could be made persistent before the first.

Persistent memory: cost of memory flushing



Rust: introduction

- Created in 2010
- 1.0 released in 2015
- Current stable version: 1.59.0
- Systems programming language
  - But with high-level programming features

Rust: rules of borrowing

- Data can only have **one owner**.
- You can borrow data:
  - Many times **immutably**,
  - Or only one time **mutably**,
- But not both! (*Aliasing XOR mutability*)

Rust: rules of borrowing

```rust
let mut a: i32 = 1;

let b: &mut i32 = &mut a;

let c: &i32 = &a;

*b = 2;
```

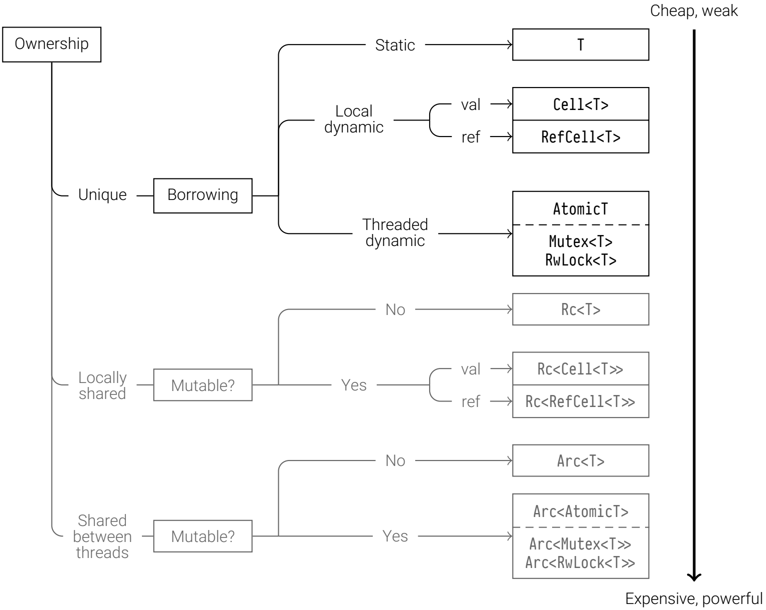
Rust: rules of borrowing

```rust
let mut a: i32 = 1;

let b: &mut i32 = &mut a;

let c: &i32 = &a;

*b = 2;
```

Rust: smart pointers

- Borrow checker sometimes too aggressive
- Cell structures & smart pointers

Rust: smart pointers



Rust: the `RefCell`

- Shifts borrow rules check from *compile time* to **run time**.
- Implemented using an internal counter.

```rust
let a = RefCell::new(0);

*a.borrow_mut() = 1;

assert!(*a.borrow() == 1);
```

`borrow_mut()` doesn't require mutability: pattern called **interior mutability**.

Our work: Taenite

- Rust library for NVRAM
- Very few dependencies
- Exploring new ways to use Rust with NVRAM
  - Fine-grained access (`MCell`)
  - Once-per-run pointer computation (Regeneration)

Programming with NVRAM: consistency

- Ensuring *logical* consistency:
  - Regular checkpointing: epochs of consistency
  - Transactions: user-defined zones (our solution)

Programming with NVRAM: mutability

- No arbitrary writes are allowed
  - All structures are read-only
  - Only allow mutation through **interior mutability**
  - All mutating functions take a journal: enforce to be in transaction

Programming with NVRAM: the `PRefCell`

- Persistent `RefCell`
- Same functionality as `RefCell` (and same cost)
  - Interior mutability using the journal

Problems with the `PRefCell`

- Update internal counter at each access: even costlier in NVRAM
- Relying on run-time correctness: bugs aren't caught at compile time

→ We can do better!

Brief introduction of the *Ghost Cell*

- Ghost Cell: separate permission from data
  - Use tokens to represent permission
- Compile-time verification (through *lifetimes*)

NVRAM: lifetimes don't make sense $\rightarrow$ need adaptation.

The `MCell`

- Persistent version of the Ghost Cell
- Used to borrow part of a collection or structure
- Can reduce size of journal backups

The `MCell`

- `MRoot`: issues tokens
- `MCell`: contains data
- `MToken`: represents access
  - `&MToken`: read-only
  - `&mut MToken`: read & write

The `MCell`: in practice

- Useful on collections: compile-time access verification
- No counter: reduce overhead of execution time & memory usage
  - Also useful to better fit in a cache line

Experimental results: the setting

Using Taenite, we made:

- Persistent hashmap library
- Blobwar game application using the persistent hash map

Goal: measure performance `MCell` vs. `PRefCell`.

Experimental results: the platform

- Grid’5000 in Grenoble: Troll cluster
- Real NVRAM DIMMs on board
- Mapped as DAX: usable as ”filesystem”

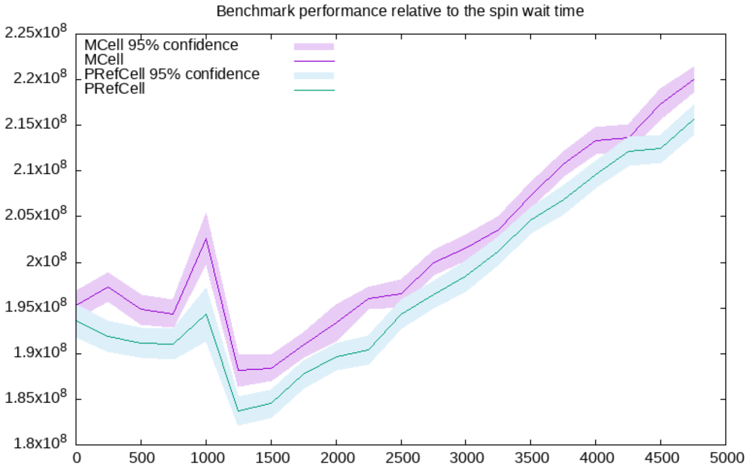
Experimental results: the benchmarks

```
for i in 0..N {
    hashmap.insert(i, i)
}
```

- Insert N elements in hashmap
- Difference: hashmap implemented with `MCell` or `PRefCell`

Experimental results: the performance analysis

Initially: performance worse (unexpected) with the `MCell`



Experimental results: the problems

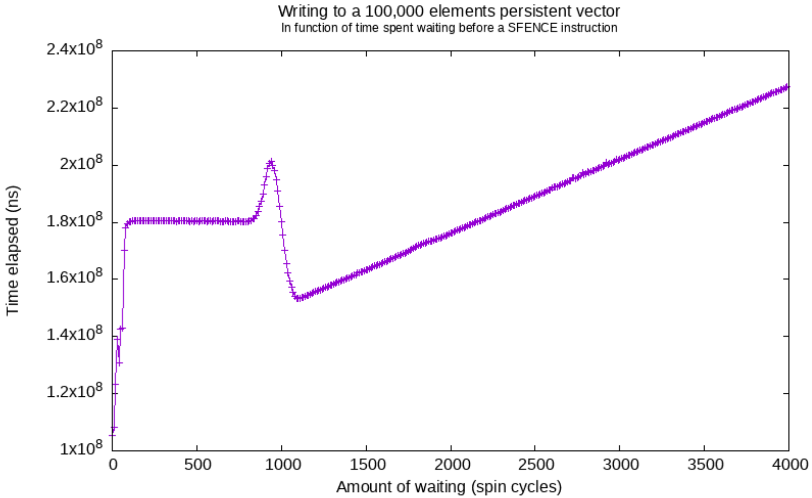
- Debugging performances on NVRAM: complicated!
  - `perf` results often misleading
  - Debugging assembly difficult
  - Online resources sparse
- What we found: time-sensitive blocking, cache alignment problems.

Experimental results: Time-sensitive delay

```
let lock = PMutex::lock();

let ptr = vec[idx].get();
*ptr += 1;
P::clflushopt(ptr);

wait(wait_cycles);

P::sfence();
lock.unlock();
```

- Varying wait time between a flush and a fence
- Quantify the time to persist data

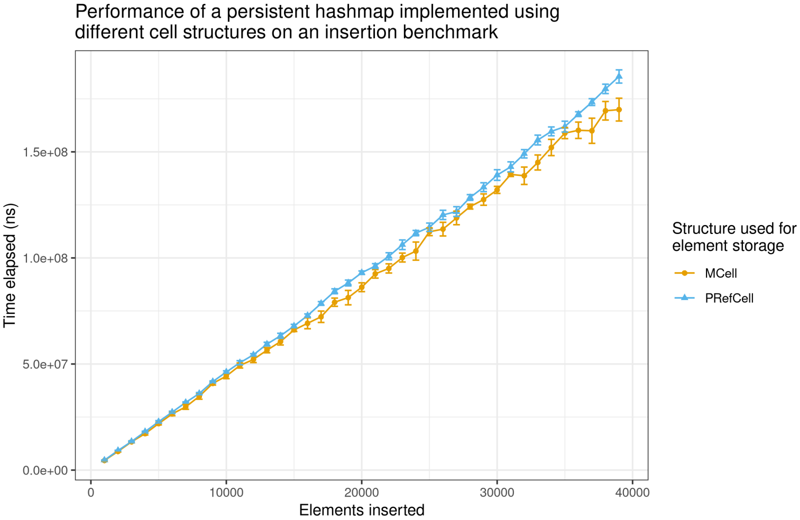
Experimental results: Time-sensitive delay



Experimental results: Cache alignment

- Flushes: work on cache line
- Align everything on cache lines
- If data overlaps: two flushes

Experimental results: Actual results



Conclusion

- Start of a library to use NVRAM in Rust
- Optimisations on performances: `MCell`, regeneration, ...
- Still ways to go: what direction?
- Strict model of behaviour of flush / fence instructions?