Carbide: A Safe Persistent Memory Multilingual Programming Framework

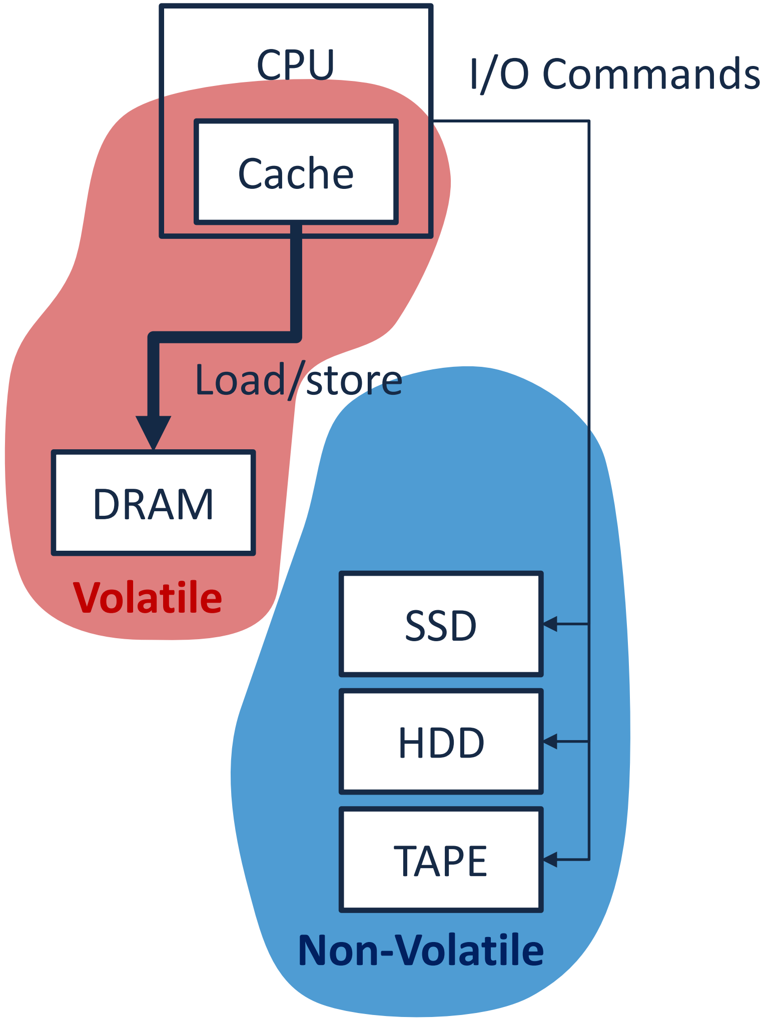
Morteza Hoseinzadeh* and Steven Swanson

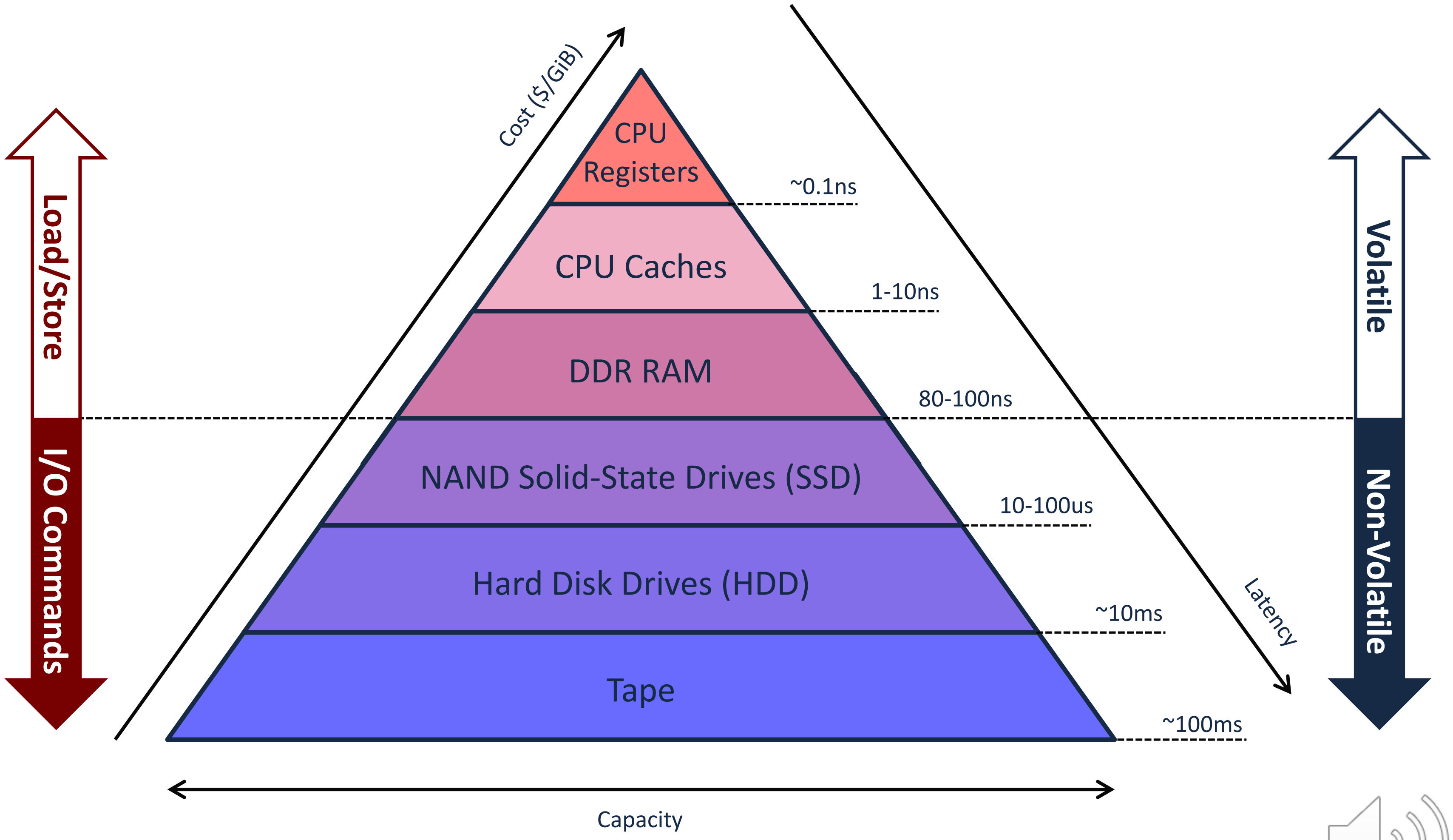
Non-Volatile Systems Laboratory
Department of Computer Science & Engineering
University of California, San Diego

** The first author is currently employed by Oracle*

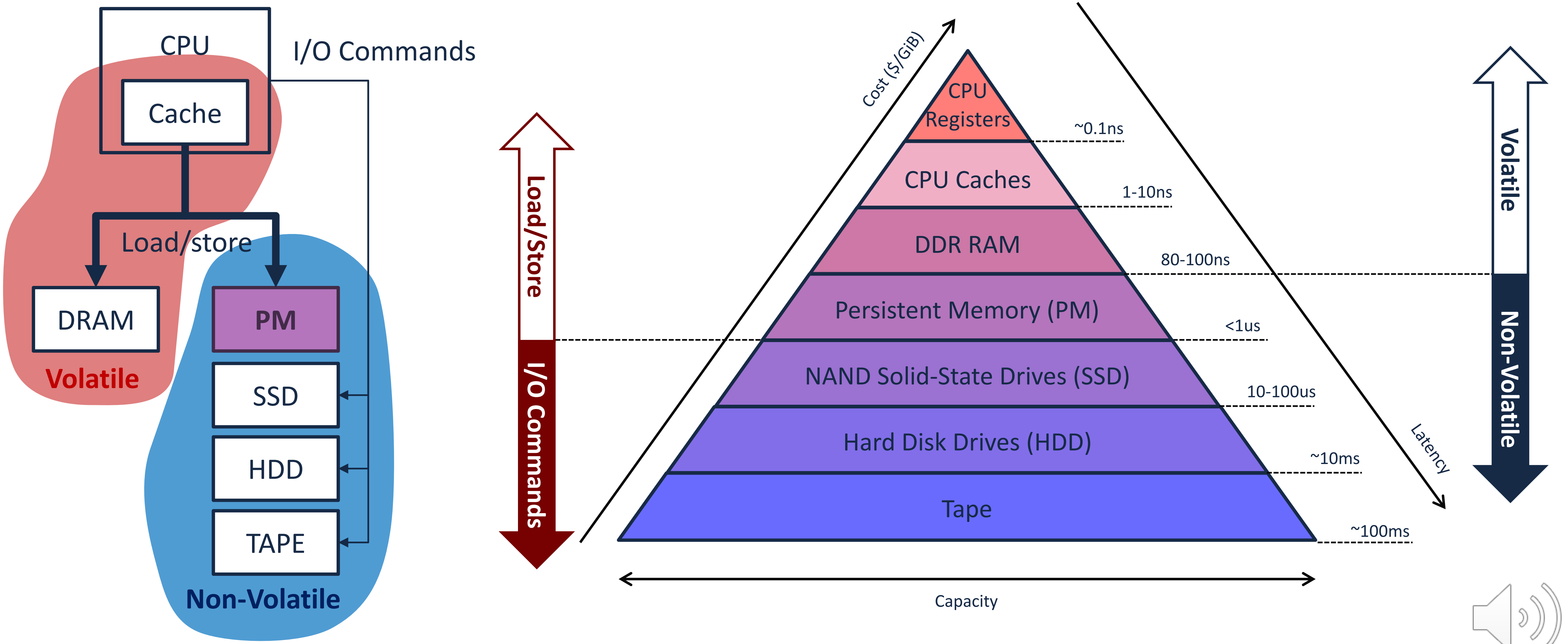
NVMW 2022

Persistent Memory (PM)





Persistent Memory (PM)

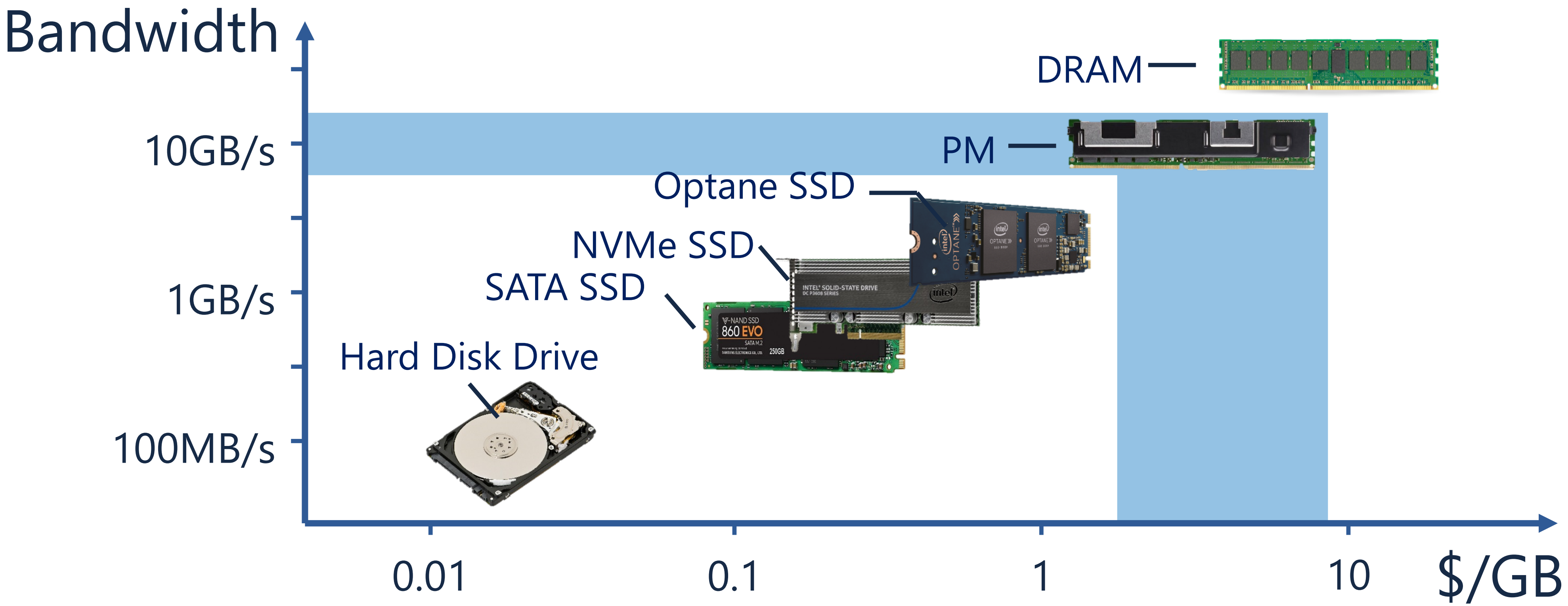


Challenges of Using PM in User-Level Applications

- Intensified current programming challenges (e.g., memory leaks)
- Persistent data consistency
  - Volatile CPU caches reorder the updates
  - No atomic compare-swap-persist instruction exists
  - Stores are not persistent until cache line is flushed
  - Non-temporal stores and cache-line flush instructions are expensive
- PM management burden is on user applications
- Handling hardware errors directly in the applications

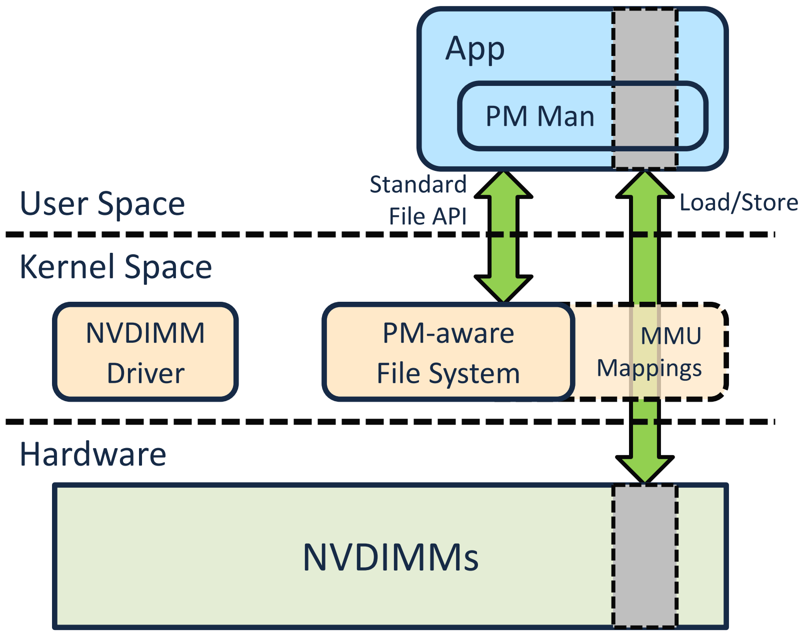
Challenges of Using PM in Storage Systems

- Inefficient usage of PM when used as a block device
- Limited scalability due to PM's expensive price



PM Programming Model

- A set of standards for enabling application development for persistent memory to address the PM programming challenges:
  - + DAX enabled file system on PM
  - + `mmap()` files (mem pools) to the virtual address
  - + User space memory management



PM Programming Frameworks

1. *Basic PM Programming Frameworks*
   - Provide interface to access PM
   - Make no safety guarantee on usage
   - Examples: PMDK, Atlas, go-pmem, Mnemosyne, and NV-Heaps

2. *Code Transformation Frameworks*
   - Statically analyze the code and inject PM operations
   - Limit the flexibility to make the program state machine smaller
   - Examples: AutoPersist, NVTraverse, Mirror, and Hippocrates

3. *Debugging/Bug-Fixing Tools*
   - Statically analyze the code and do symbolic execution to find the bugs
   - NP-Hard problem, and path explosion in large programs
   - Examples: NVL-C, Jaaru, and Agamotto

PM Programming Frameworks

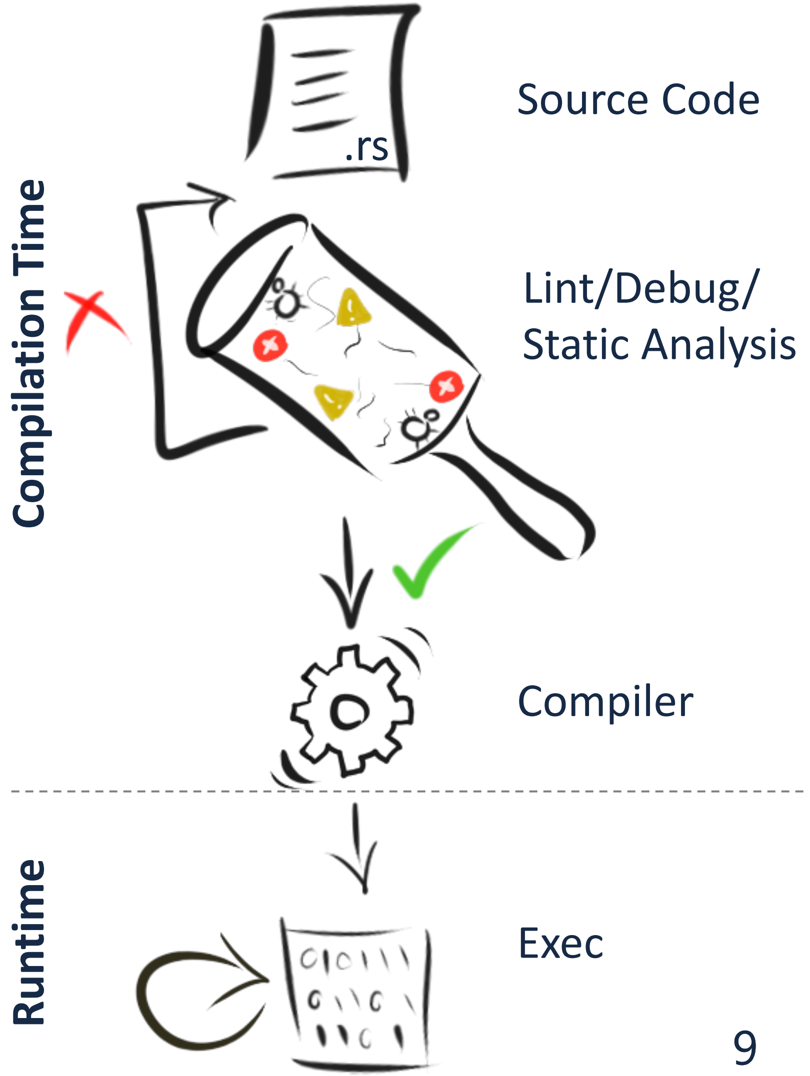
4. *Testing Frameworks*
   - Dynamically inject failures to test the program
   - Completeness proof is not provided
   - Examples: PMTest and XFDetector

5. *Pre-Compilation Debugging Frameworks*
   - Apply safety rules statically as it's being developed
   - Limit the flexibility as they apply restrictive safety rules
   - Example: Corundum

Corundum

- A PM programming library for Rust
- Enforces PM safety at compile time
- High performance due to static analysis
- Idiomatic approach to support PM
- It guarantees no PM-related bug

PM Safety ⊂ Rust's Type Safety



Corundum Challenges

- Too restrictive
- Risky optimizations are not possible
- Steep learning curve for non-Rust developers

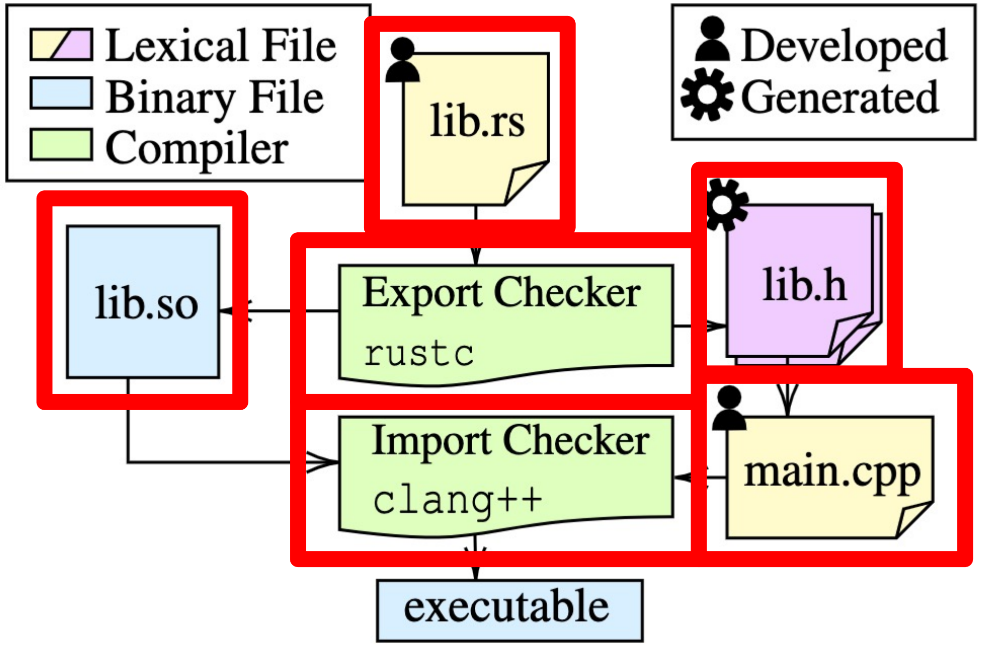
CARBIDE:

USE CORUNDUM FOR DEFINING PERSISTENT TYPES

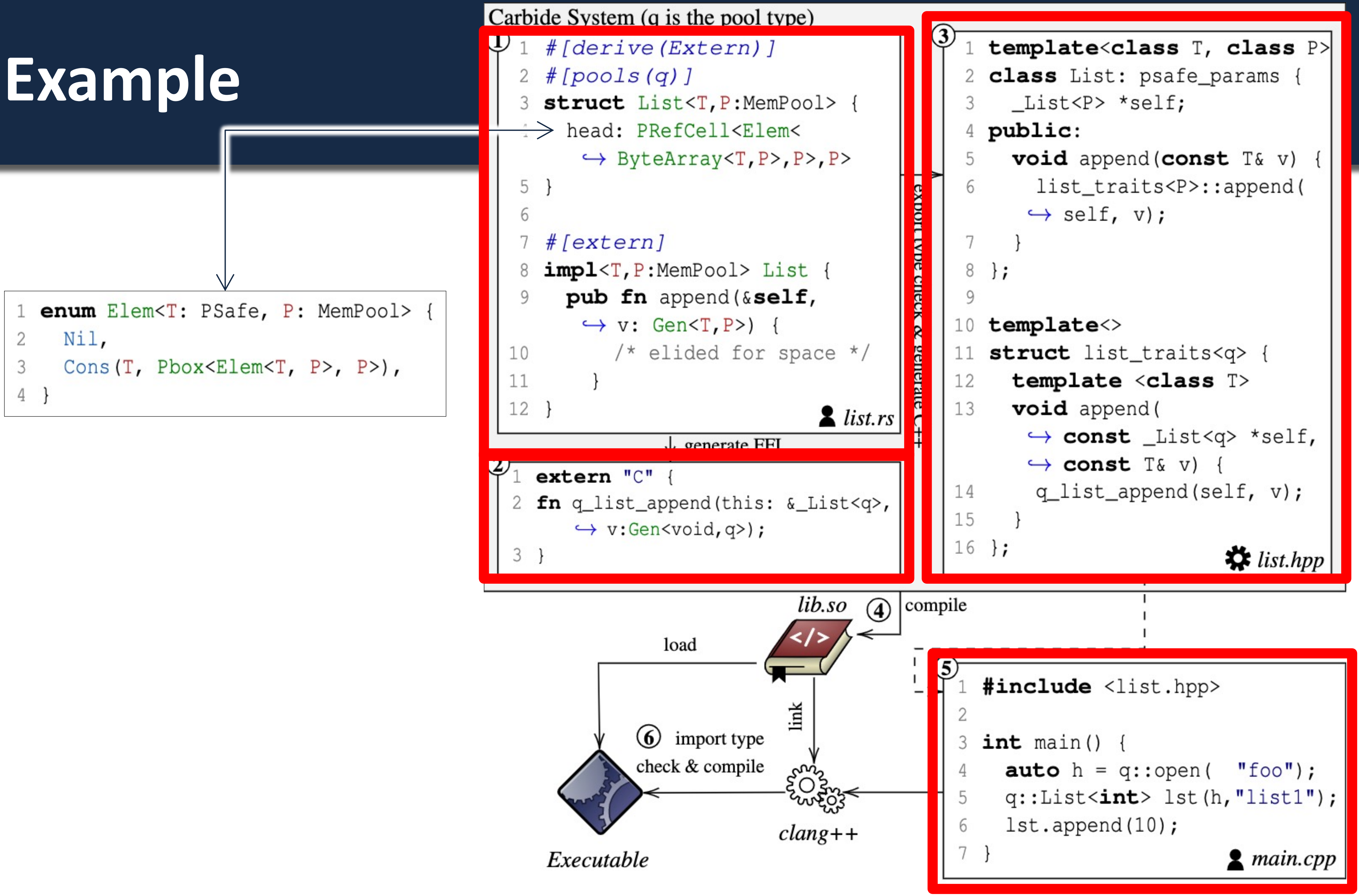
USE C++ FOR DEVELOPING THE PROGRAM

Carbide

- Developing persistent data structure type separately using Corundum in Rust (*lib.rs*)
- Strict rules apply to persistent types only
- Data types are externally available through a dynamic library (*lib.so*) with an automatically generated **API** (*lib.h*)
- The Export Checker statically checks the container types for the capability of external usage
- The Import Checker statically checks the types being stored in PM



Example



enum Elem<T: PSafe, P: MemPool> {
Nil,
Cons(T, Pbox<Elem<T, P>, P>),
}
Carbide System (q is the pool type)
1
#[derive(Extern)]
#[pools(q)]
struct List<T,P:MemPool> {
head: PRefCell<Elem<
↪ ByteArray<T,P>,P>,P>
}
#[extern]
impl<T,P:MemPool> List {
pub fn append(&self,
↪ v: Gen<T,P>) {
/* elided for space */
}
}
list.rs
generate FFI
export type check & generate C++
2
extern "C" {
fn q_list_append(this: &_List<q>,
↪ v:Gen<void,q>);
}
3
template<class T, class P>
class List: psafe_params {
_List<P> *self;
public:
void append(const T& v) {
list_traits<P>::append(
↪ self, v);
}
};
template<>
struct list_traits<q> {
template <class T>
void append(
↪ const _List<q> *self,
↪ const T& v) {
q_list_append(self, v);
}
};
list.hpp
lib.so
4
compile
load
link
6
import type
check & compile
clang++
Executable
5
#include <list.hpp>
int main() {
auto h = q::open("foo");
q::List<int> lst(h,"list1");
lst.append(10);
}
main.cpp

Carbide's Design Goals

API Design

1. Preserve the same guarantees as Corundum's
2. Provide a seamless cross-language PM management system
3. Provide a safe C++ interface to interact with data as defined in Rust
4. Statically checked the external usage of the persistent type definition in Rust
5. Statically checked the usage of external persistent type declaration in C++
6. Specify a design pattern to make a C++ type persistent

API Design Challenges

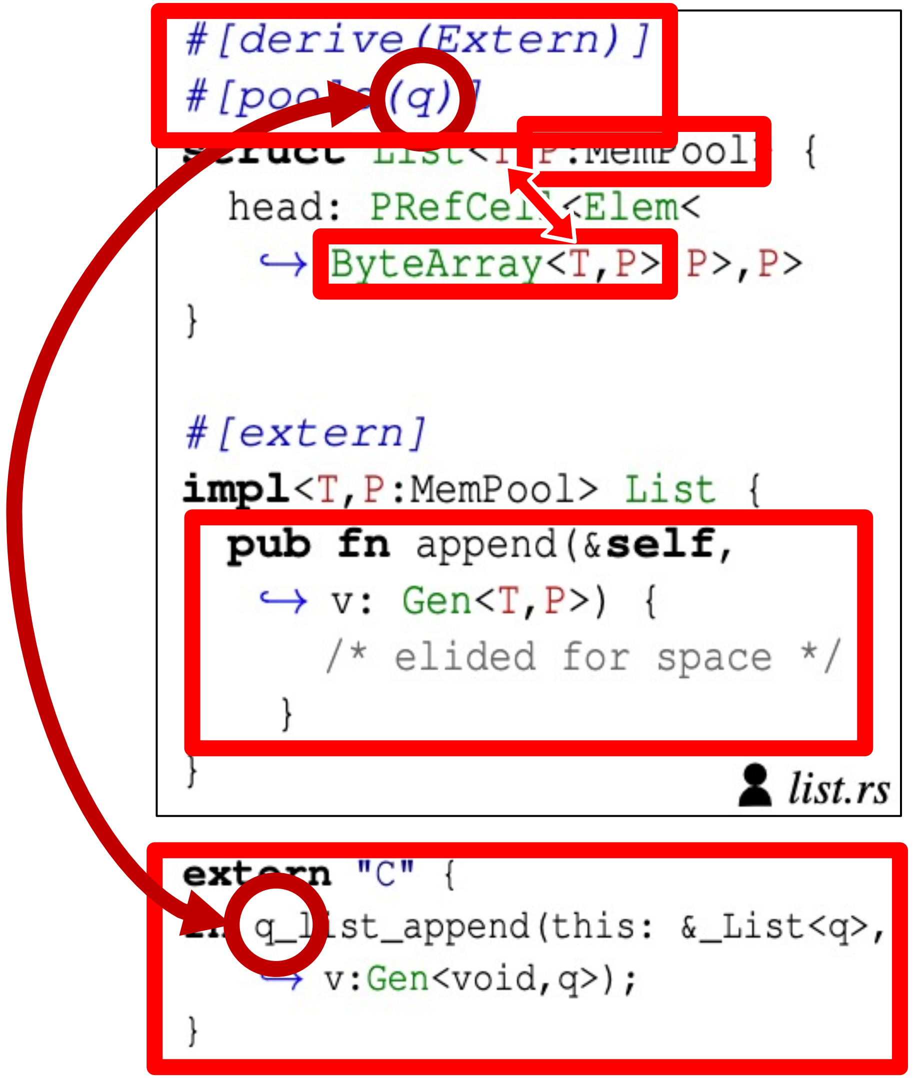
- Type Interoperability
  - Rust and C++ layout memory differently
- Polymorphism
  - Polymorphic types are not available through dynamic libraries in Rust and C++
- Memory Leaks
  - C++ does not garbage collect when dynamic allocation is used
- Lifetime Conflict
  - The RAII model in C++ and Rust have distinct lifetime scopes

Type Interoperability

- Portable type:
  - Annotate external types for a specific set of pools
  - Corundum's rules apply (Rust's type system)
  - Exactly one pool type parameter
  - Other type parameters are used in form of byte arrays
  - External interface is FFI-compatible
  - Provide at least one transactional constructor

Export Rules

- Carbide exports the type's functionality by generating an FFI for every specified **pool**



Polymorphism

- Type-parameter reduction and reparameterization
  - Specialize the data type parameters with **void**
  - Specialize the pool type parameters for every specified pools and generate the FFIs
  - Implement a C++ template class (vessel class) with the same parameters and functionality
  - Implement the type traits for the given pools in C++ to call the corresponding APIs

```
template<class T, class P>
class List: psafe_params {
  _List<P> *self;
public:
  void append(const T& v) {
    list_traits<P>::append(
    ↪ self, v);
  }
};
```

```
template<>
struct list_traits<q> {
  template <class T>
  void append(
   ↪ const _List<q> *self,
   ↪ const T& v) {
    q_list_append(self, v);
  }
};
```

list.hpp

Memory Leaks

- There is no PM dynamic allocation available in C++
- Only Carbide’s internal types can manage PM
- Every allocation is owned by an object in Rust

Lifetime Conflict

- Lifetime of a C++ object is unknown when passed to a foreign function
- The object's resources are release at the end of the scope

C++

```
{
  Obj a; // constructor call
  foo_ffi(a);
} // a.~Obj() destructor call

{
  Obj *b = bar_ffi();
  read(b); // access violation
}
```

Rust

```
Obj b; // never drops

fn foo_ffi(a: Obj) {
  b = a;
}

fn bar_ffi() -> &Obj {
  &b
}
```

Extended RAII

- A hyper scope is a scope extending from C++ to Rust
- **`Gen<T,P>`** as a cross-language reference type lives in a hyperscope
  - Defined in both Rust and C++
  - Contains a ***relative pointer to the destructor*** function to call from Rust
  - Dynamically allocates and construct the object when instantiated in C++
  - Does not immediately release resources in the destructor
  - Can merely move the resource to a **`ByteArray<T,P>`**

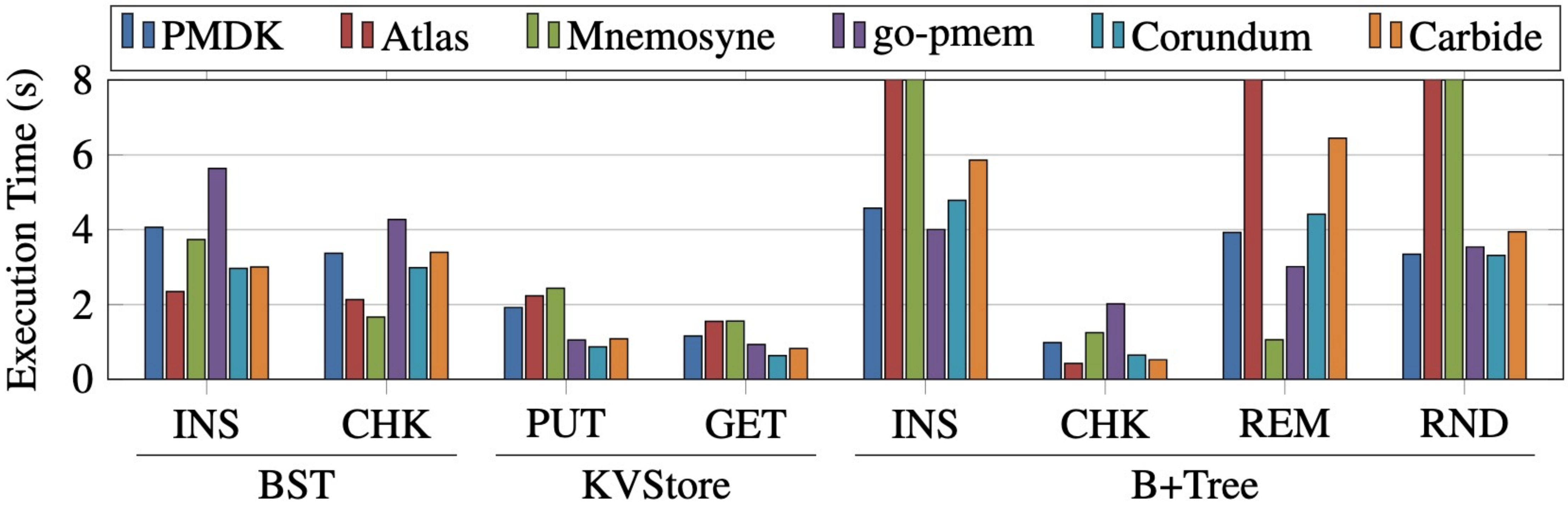
C++

```
------------------------------------
TXN {
  Gen<Obj> a; // dyn alloc, cons
  foo_ffi(a);
}

{
  Obj *b = bar_ffi();

  read(b); // Ok
}
```

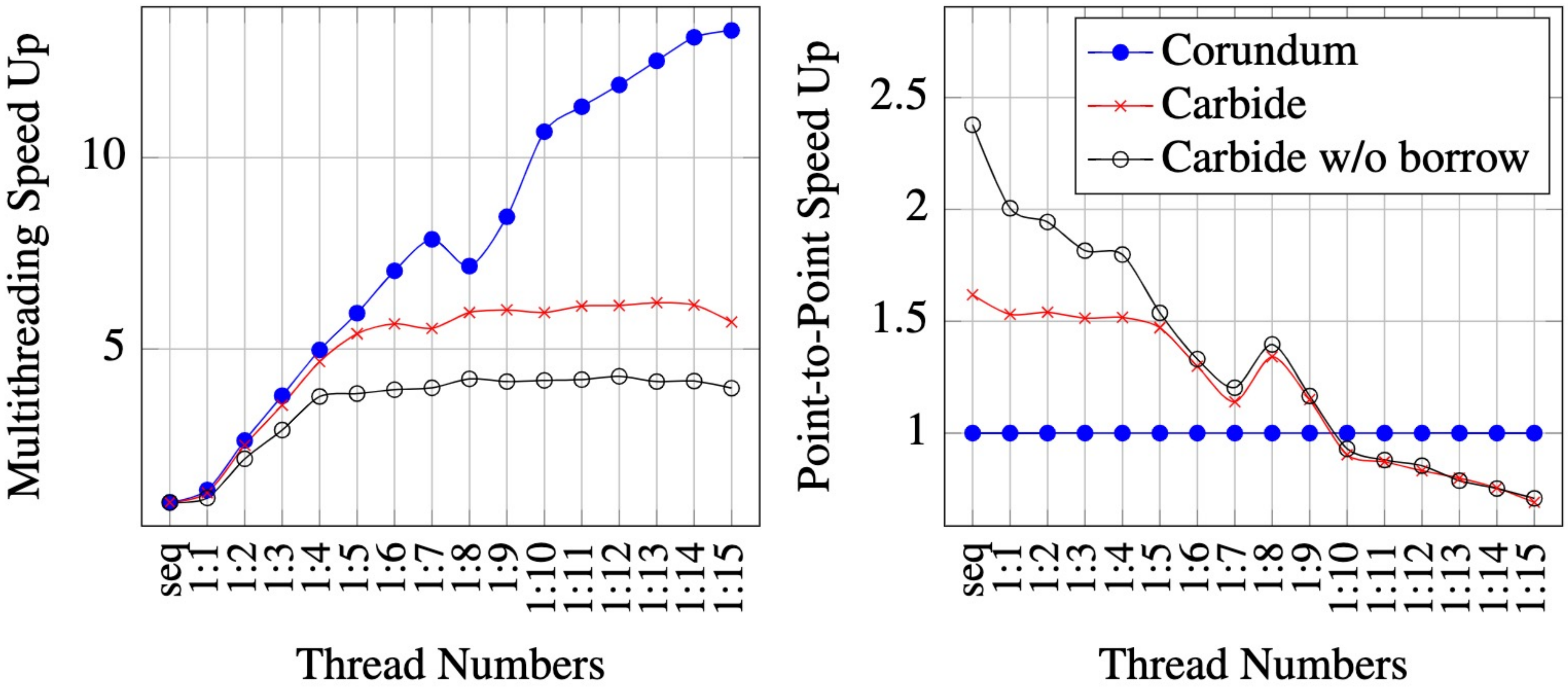
Rust

```
-----------------------------------
ByteArray<Obj> b; // never drops

fn foo_ffi(a: Gen<Obj>) {
  b = ByteArray::from(a);
}

fn bar_ffi() -> Gen<Obj> {
  b.as_gen()
}
```

Performance Results



Optimization Impact and Scalability



Conclusion

- PM is an advanced memory technology that offers both high-performance and non-volatility
- PM programmers face a set of safety challenges, as well as higher price per GB compared to other NVM block devices
- Current PM programming frameworks exclusively offer safety or programming flexibility
- We presented Carbide, a PM programming framework that allows using Corundum data structures in C++ to guarantee safety as well as programming flexibility

Thank you!