Carbide: A Safe Persistent Memory Multilingual Programming Framework

1 Motivation

Persistent memory (PM) technology has brought the opportunity of accessing persistent data directly by using load and store instructions, improved memory system capacity, and a unified programming model for persistent and volatile programs. Many PM libraries [1–6] have been introduced to operate on PM devices safely. Although they provide safe interfaces to interact with PM, not all of them verifiably address novel PM programming challenges. Of a particular class of PM libraries, some of them provide strong guarantees to prevent PM-related bugs. Corundum [4] is the state-of-the-art in this category, but it applies several restrictive rules that thwart performance optimizations. The diversity in the offerings and limitations of the current PM programming frameworks motivates us to look for a hybrid PM programming model that can be both flexible and verifiably safe.

2 The Key Insight

Our work relies on the fact that even though the persistent data types should be implemented strictly safely, the other parts can be developed more freely. Hence, we propose that using Corundum [4] to implement persistent types and port them into C++ in the form of a compiled library can provide safety and flexibility at the same time.

3 Main Artifacts

This paper presents Carbide, a multilingual PM framework that allows separately developing PM data structures in Corundum and using them in C++. This improves the flexibility in programming and code reuse while maintaining strong PM safety guarantees. Figure 1 depicts Carbide's system description. Multilingual programming essentially requires the programmer to consider strict directives to follow. Carbide's automatic code generation and static type checkers ensure that the safety invariants are satisfied in both languages. Our contributions in developing the Carbide framework is:

- Carbide preserves almost all Corundum's safety guarantees in C++.
- Carbide introduces a notion of the expanded lifetime of persistent objects with lifespans stretching between Rust's and C++'s scopes.

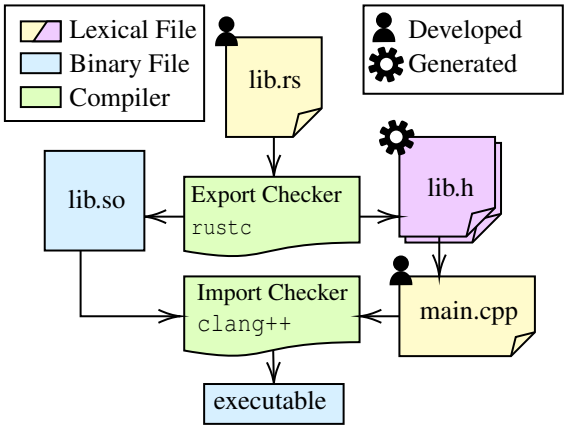

Figure 1: Carbide's System Description. The programmer defines persistent data types in *lib.rs* using Rust; Carbide generates the library and the header files; The programmer uses them in *main.cpp* using C++.

- Carbide statically checks the exporting procedure to prevent unsafe access to persistent data in Rust.
- Carbide performs static type checking while externally using the persistent objects from C++.
- Carbide transfers polymorphic types from a compiled library through a type parameter reduction and reparameterization technique.
- Carbide provides an option for automatically converting volatile data structures, including the C++ standard template library's container types, into persistence under specific criteria.

4 Key Components

Implementing such a system may jeopardize the safety guarantees that Corundum provides because many of the techniques used in Corundum are based on Rust's features, and they are not available in C++. Below we list the main safety concerns and our solutions for them:

Type Interoperability The types implemented in Rust have different memory layouts from C++; therefore, they are

not fully operable. Rust allows the programmer to specify a "C" representation of the type explicitly, but this is only available for trivial types such as a struct with primitive fields. The persistent pointers and type wrappers are not specified so. Carbide automatically generates a C++ interface for every Corundum type as a vessel to make the type fully operational in C++.

Polymorphism Persistent types are usually defined as container classes that take on many forms. However, a polymorphic type is not portable through a compiled library as the type parameters cannot be derived in post-compilation time. Carbide enforces using byte-arrays to represent any sized types and carefully evaluate them to prevent any misinterpretations.

Memory Leaks C++ is not a memory-managed programming language; dynamic allocations may remain unreachable and unclaimed, leading to memory leaks. In Carbide, a static checker guarantees that all persistent objects have an owner in Rust that manage their allocations.

Conflicted Lifetimes Although both C++ and Rust use an RAII model, the two programming languages have different scopes, and one object cannot be accurately managed if it travels between the two scopes. Carbide redefines RAII model that extends an object's lifetime from C++ to Rust.

Our experiments show that Carbide vastly improves programmability at the cost of zero or small performance overhead when the same design is used. It improves performance by up to 60% when performance optimizations apply.

5 Key Results and Contributions

Figure 2 shows our experimental results comparing Carbide's performance with PMDK, Atlas, Mnemosyne, go-pmem, and Corundum. Carbide performance is comparable to other PM systems according. In BST and KVStore workloads that do not use unsafe coding and objects do not move very often, Carbide performs almost similar to Corundum. In the B+Tree workloads, multiple factors adversely affect Carbide's performance: 1) data copies around several times and each copy needs double allocations for the reference and the byte array, 2) similarly, dropping objects also calls deallocation twice, and 3) the byte array type has a drop function even if the underlying data type does not have any destructor (e.g., primitive types). Overall, the results confirm that Carbide's hybrid programming framework adds Corundum's safety to the flexible programming environment in C++ without incurring a great deal of performance overhead.

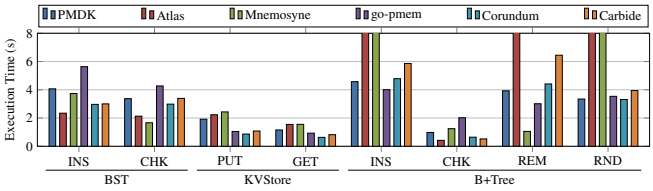

Figure 2: The performance results of comparing Carbide with other PM systems in terms of execution time.

References

[1] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[2] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[3] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 859–872, 2020.

[4] Morteza Hoseinzadeh and Steven Swanson. Corundum: Statically-enforced persistent memory safety. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, page 429–442, New York, NY, USA, 2021. Association for Computing Machinery.

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

[6] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.