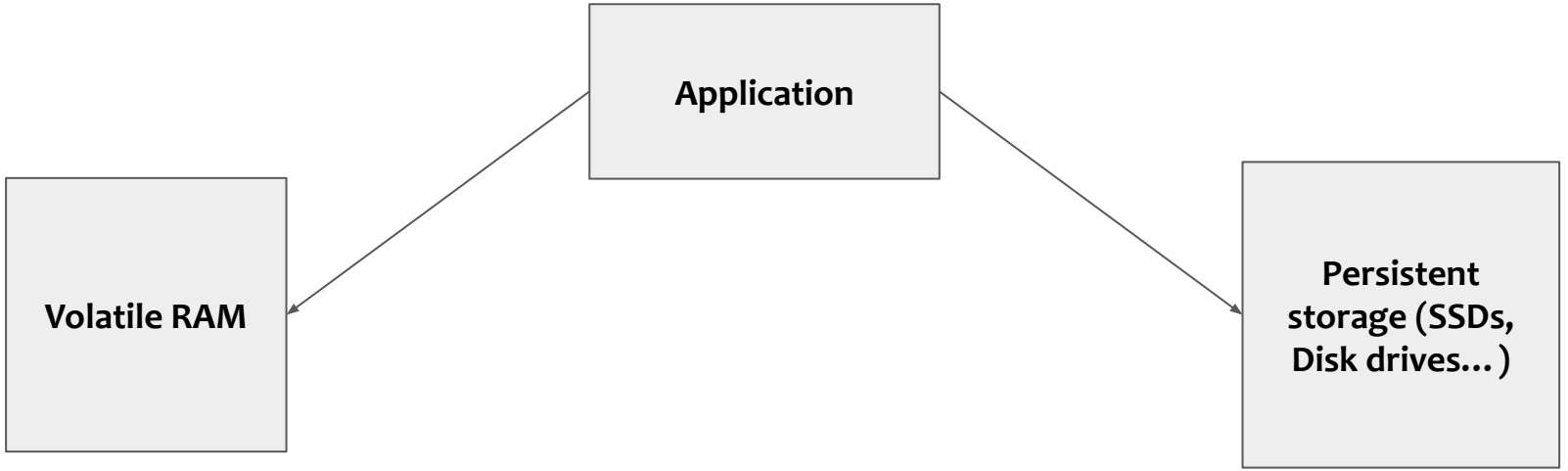
Bachelor's thesis

Programming model for hybrid persistent memory systems

Matthias Werndle
Advisor: Dimitrios Stavrakakis
Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

15.04.2022 – 16.08.2022

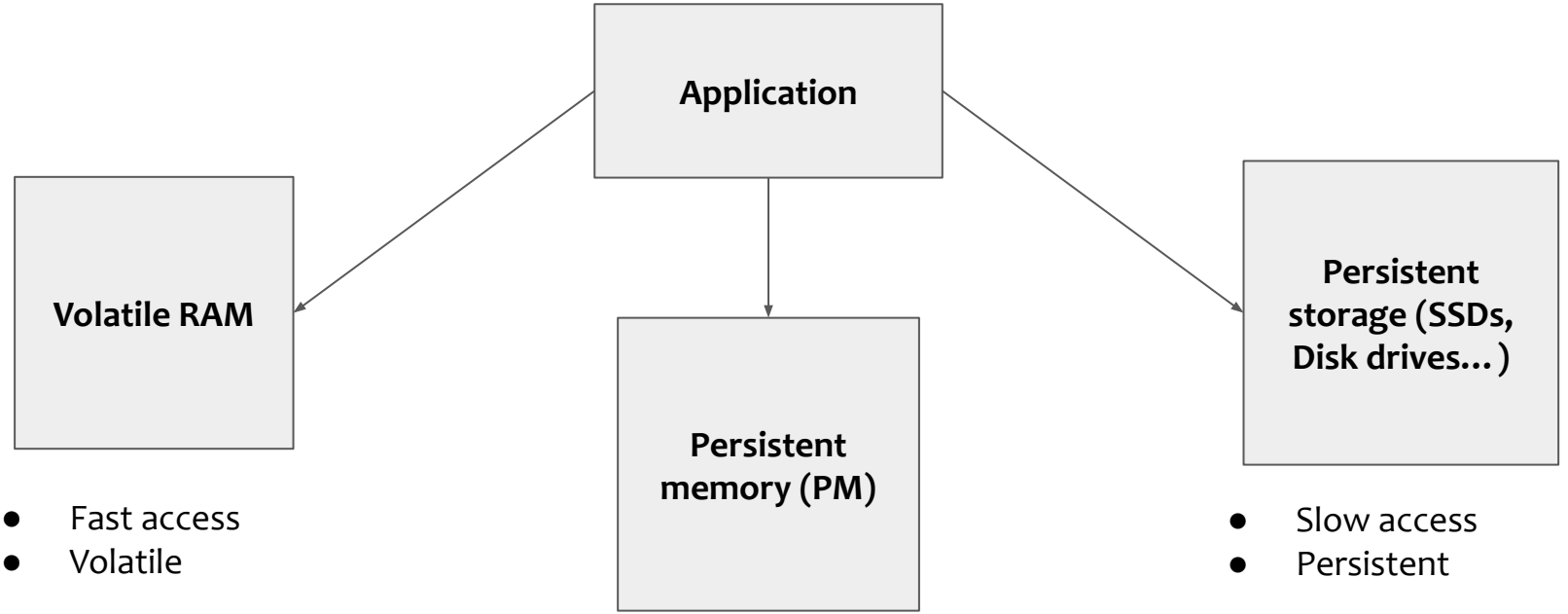
Introduction



- Fast access
- Volatile

- Slow access
- Persistent

Introduction



- Fast access
- Volatile

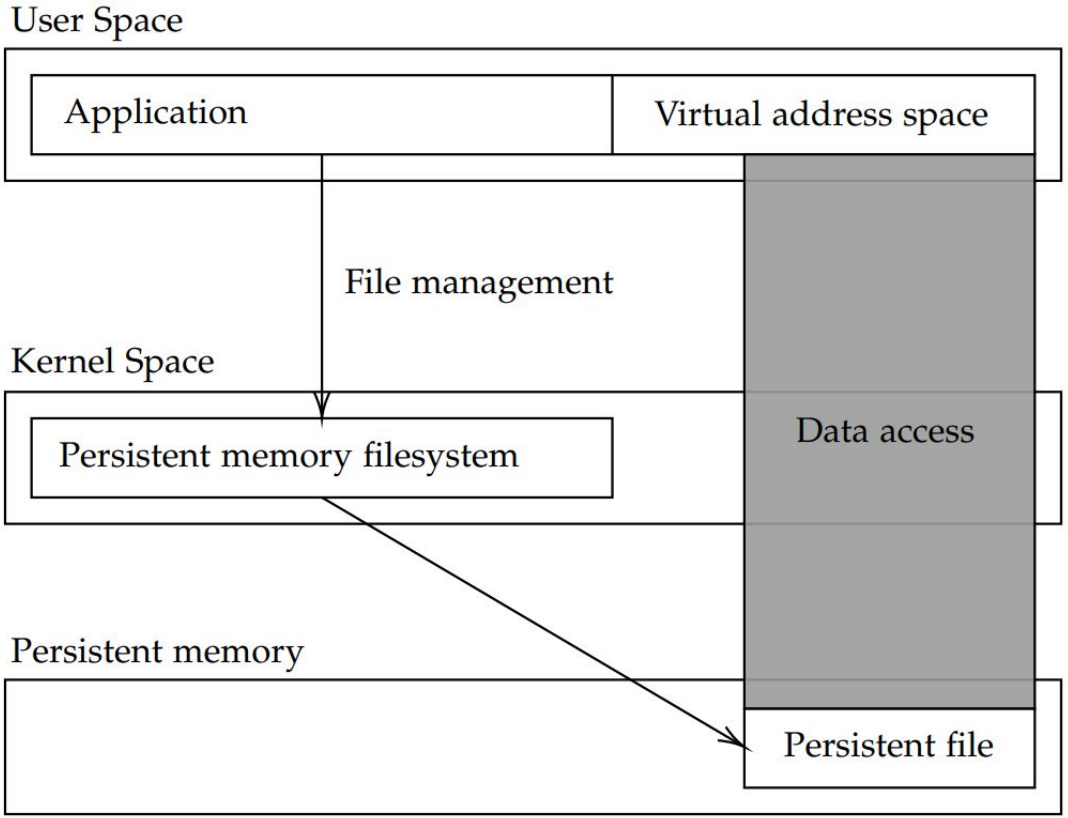
- Slow access
- Persistent

- Access latency close to DRAM
- Persistent

Background - Programming model

- PM space is structured as files or regions
- Kernel handles PM file access
- Memory mapping allows direct access to PM file

⇨ **Pointer relocation requires special handling**

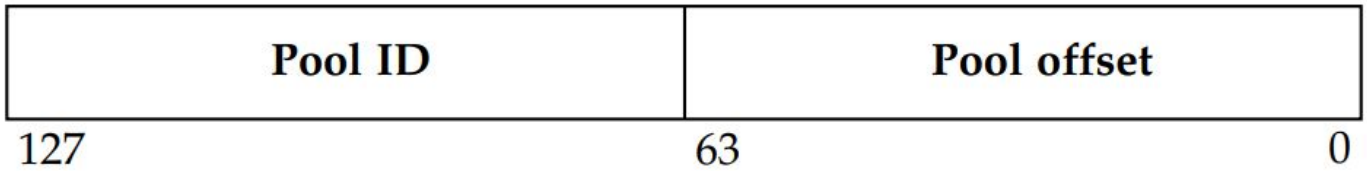


Background - PMDK

Two pointer types:

- Volatile Memory (**VM**) Pointers
- Persistent Memory (**PM**) Pointers

PM Pointer layout:

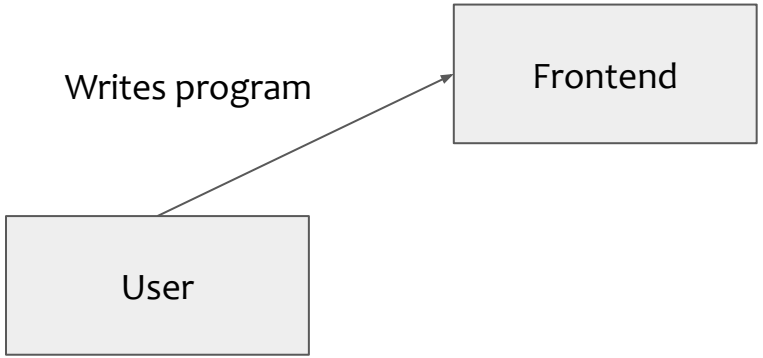


- PM pointer is **independent** of memory mapped address space
- Requires conversion from **PM** pointer to **VM** pointer to access data
- Requires **additional** data types

Current issues

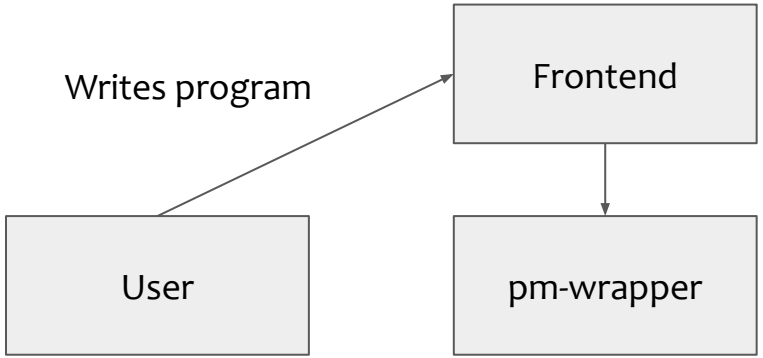
- **Different data structures** are required for PM support depending on the used library
- It is hard to **convert** existing **VM** applications into **PM** applications
- Programs are **depending** heavily on one library

Design - Overview



➔ Allows pointer access to persistent memory like with C pointers

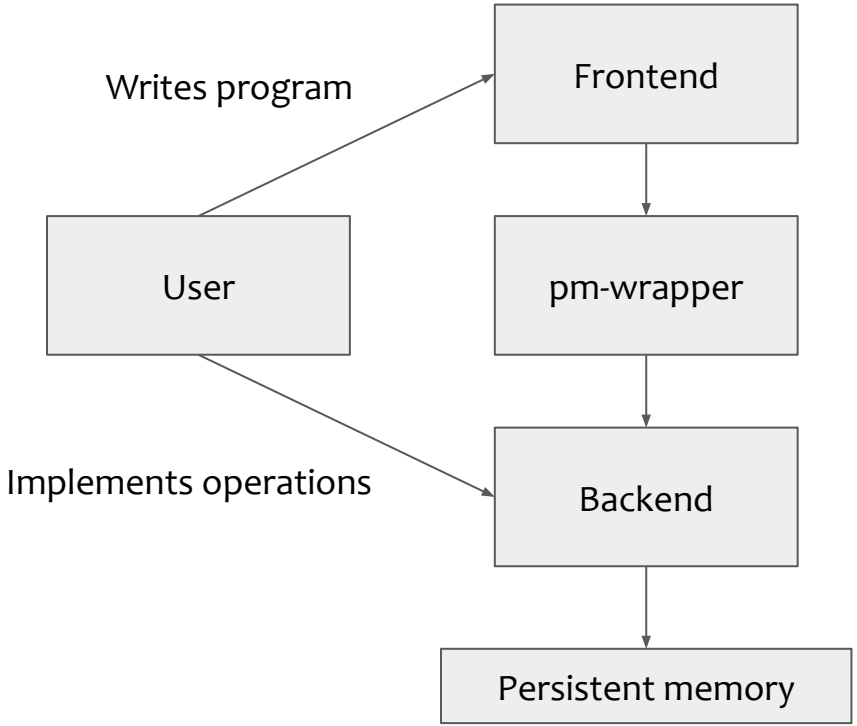
Design - Overview



➔ Allows transparent c like pointer access to persistent memory

➔ Layer which consumes source code to perform PM operations

Design - Overview



- ➔ Allows transparent c like pointer access to persistent memory
- ➔ Layer which consumes source code to perform PM operations
- ➔ Manages connection to PM
- ➔ Implementation can use library such as PMDK

Design - Frontend

- Introducing a new implicit type system to differentiate between pointer types

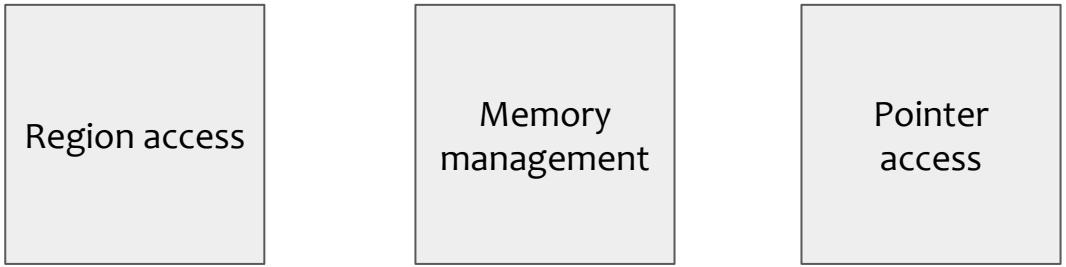
```
1  // VM pointer
2  int* vm_ptr = malloc(...);
3
4  // PM pointer
5  int* pm_ptr = pm_alloc(...);
```

- Usual pointer operations can be performed on PM pointers
- Undetectable pointer types have to use attributes

Overall frontend allows PM access similar to VM programs

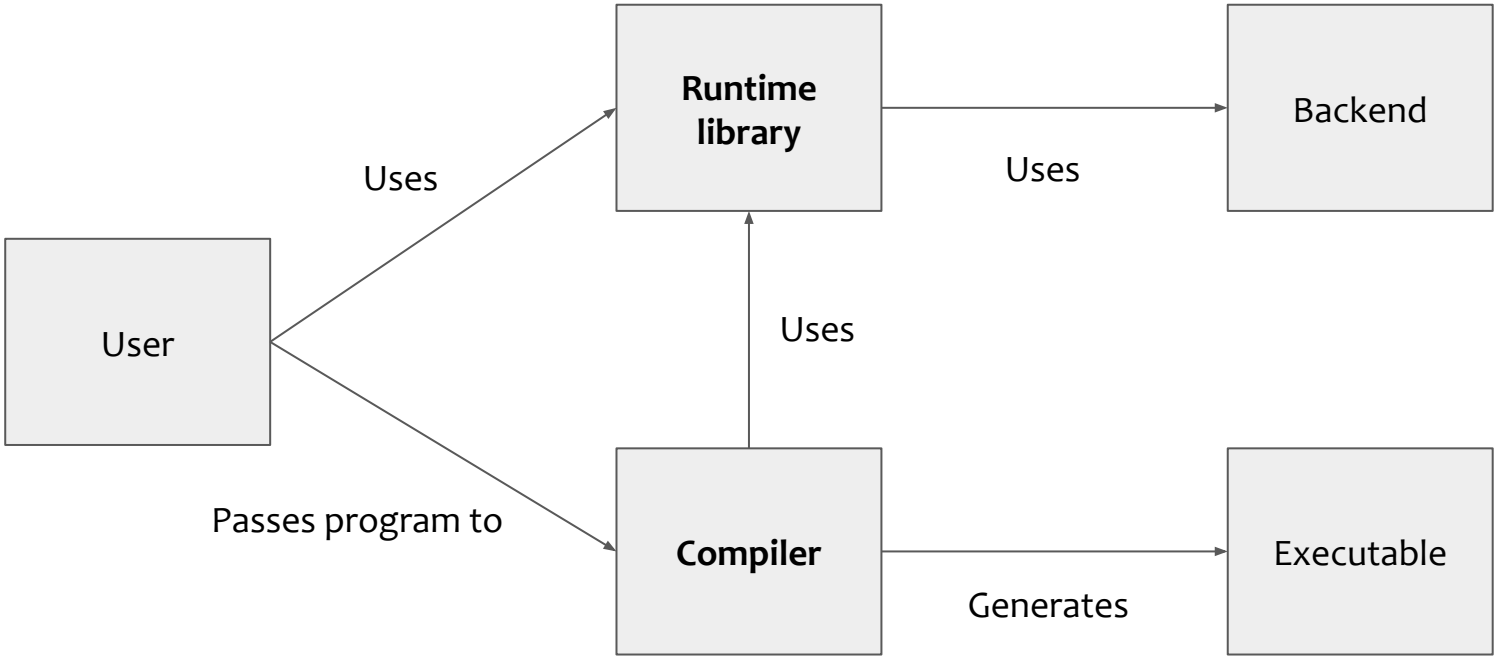
Design - Backend

- Handling low level PM access
- Operations types:



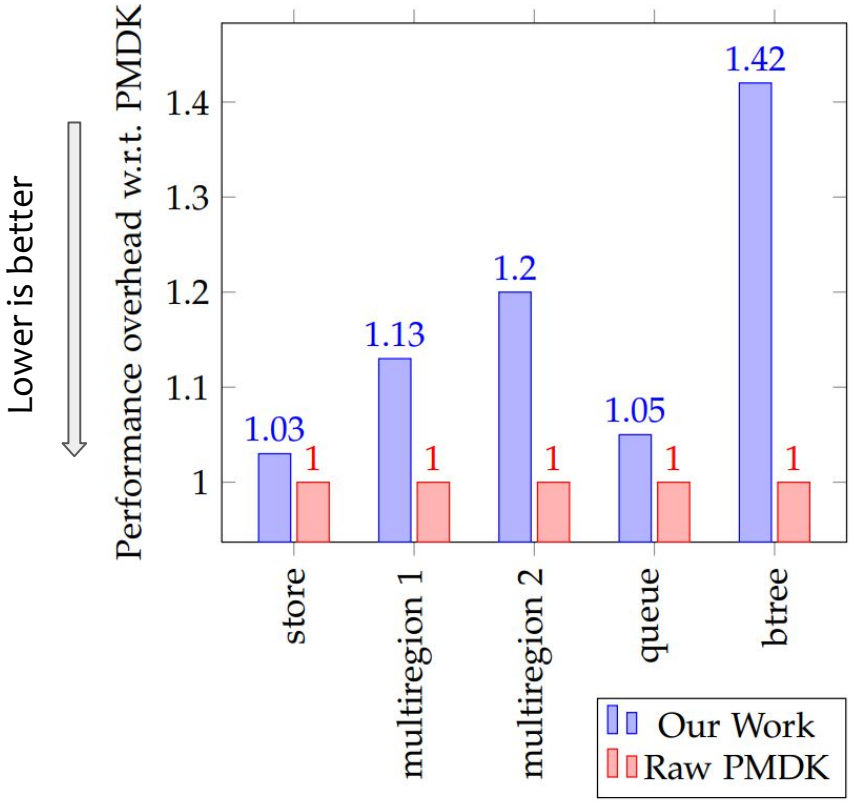
Backend allows to switch implementations without changing the frontend

Implementation - Overview



Two major parts: **Runtime library** and **Compiler**

Evaluation - Runtime overhead



- Compares raw PMDK versions with pm-wrapper versions implemented with PMDK
- Compiled with Clang-13 and O3 optimization

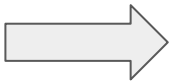
Overall low performance overhead

In some cases compiler compiler inserts have to be optimized (btree)

Evaluation - Program migration

Migration of a VM hashtable to a PM hashtable

Change	Description	Count
Function relacements	Switching a called function with another	6
Structure changes	Adding, removing or altering members of structs	0
Additional logic	Inserting additional required logic	4
PMI type attributes	Required type attributes	5
Pointer operation	Required pointer operations changes	0



Migration is improved, but in some cases the compiler still requires type attributes

Conclusion

➔ Quicker migration to PM support

➔ Simpler migration between PM libraries

➔ Performance overhead is low

pm-wrapper provides an additional layer to improve PM programming

Source Code: https://github.com/Teppichseite/pm-wrapper

Backup

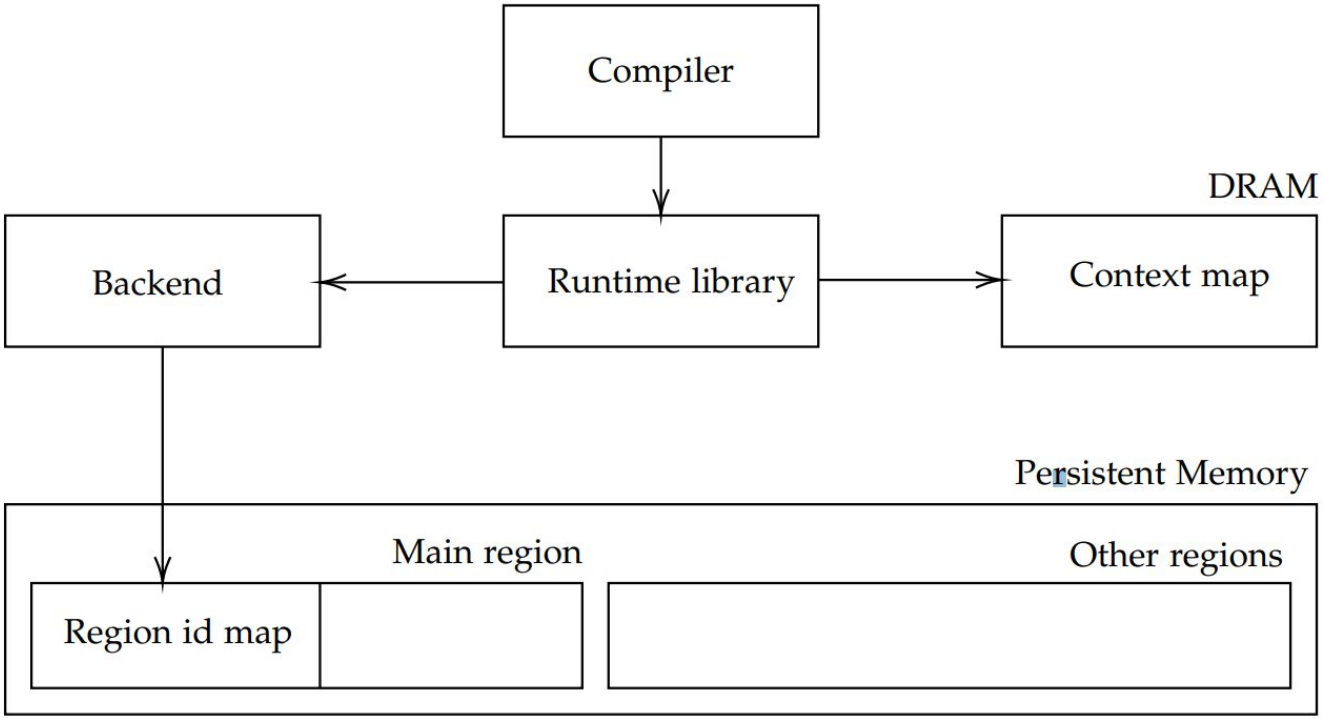
Design - Frontend example

```
1  struct Store {
2    int *ptr_a;
3    int *ptr_b;
4  };
5
6  struct Store *store1 = (struct Store *)pm_alloc(sizeof(struct Store));
7
8  // PMI type PM
9  int *ptr1 = store1->ptr_a;
10
11 struct Store *store2 = (struct Store *)malloc(sizeof(struct Store));
12
13 // PMI type PM by attribute
14 PM int *ptr2 = store2->ptr_a;
15
16 // PMI type VM
17 int *ptr3 = store2->ptr_b;
```

Design - Backend struct

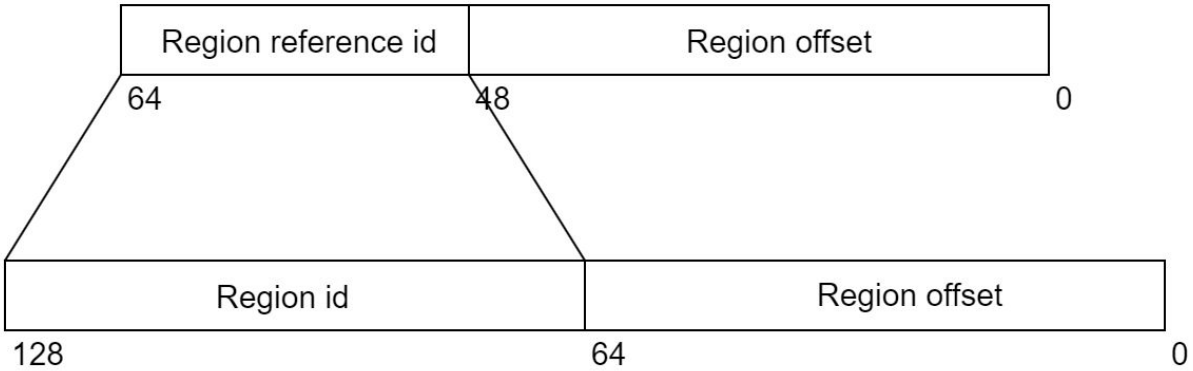
```
1  struct PmBackend {
2    int (*init)();
3    int (*open_or_create)(PmBackendContext *context, bool *created_new);
4    void (*close)(PmBackendContext *context);
5    void (*finalize)();
6    pm_region_offset (*get_root)(PmBackendContext *context);
7    pm_region_offset (*alloc)(PmBackendContext *context, size_t size);
8    pm_region_offset (*calloc)(PmBackendContext *context, size_t size);
9    void (*free)(PmBackendContext *context, pm_region_offset offset);
10   void *(*read_object)(PmBackendContext *context, pm_region_offset offset);
11   void (*write_object)(PmBackendContext *context, ...);
12 };
```

Implementation - Runtime



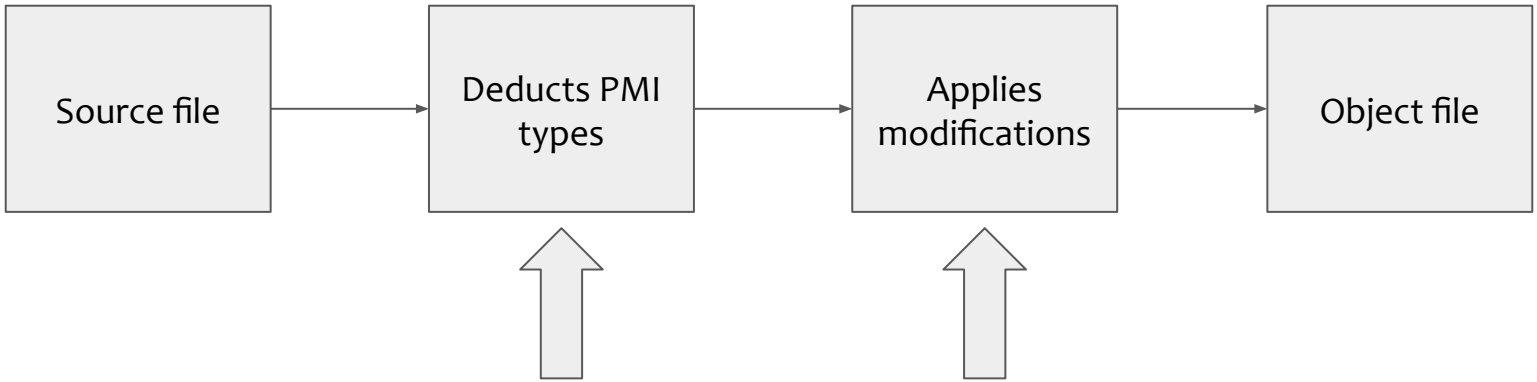
Implementation - Runtime

- Offers **region** and **memory management** interface
- Handles **mapping** between **frontend pointers** and backend **persistent pointers**
- Mapping can be realized in two ways:
  - Hash map
  - Fixed size array lookup



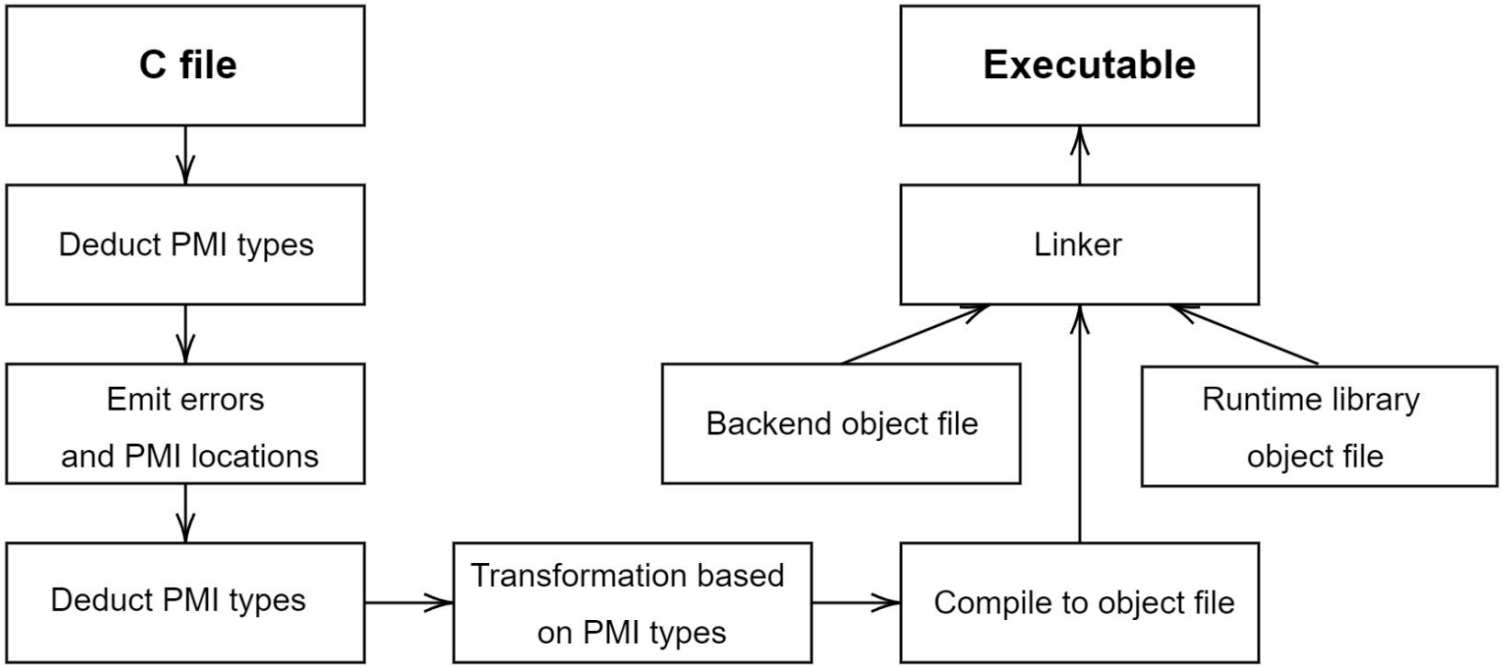
Implementation - Compiler

- Implemented with Clang LibTooling
- Analyzes directly the C source code
- Deducts pointer types
- Rewrites a copy of the actual source code



Main implementation effort for pm-wrapper

Implementation - Compiler chain



Future work - Transactions

Backend implementation

```
1 void write_object(void *dst, char *data, size_t len) {
2   // Perform logging
3   pmemobj_tx_add_range(dst, 0, len);
4   // Copy data
5   pmemobj_memcpy(dst, data, len);
6 }
```

Frontend usage

```
1 PM struct Data *data;
2 pm_tx_start();
3 data->a = 1;
4 data->b = 2;
5 pm_tx_end();
```