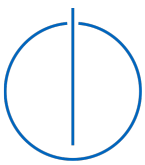
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Programming model for hybrid persistent memory systems

Matthias Werndle



DEPARTMENT OF INFORMATICS

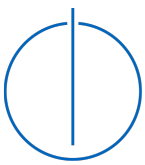
TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Programming model for hybrid persistent memory systems

Programmiermodell für hybride persistente Speicherungssysteme.

Author:	Matthias Werndle
Supervisor:	Prof. Pramod Bhatotia
Advisor:	Dimitrios Stavrakakis
Submission Date:	16.08.2022



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.08.2022 Matthias Werndle

Acknowledgments

I want to thank my advisor Dimitrios Stavrakakis who gave me always plenty of helpful advice for the general research, the implementation and the thesis documentation.

Abstract

Persistent memory allows programs to perform operations on non volatile memory which has lower latency than NAND SSDs. The programmer can interact with this kind of memory via different libraries. Persistent memory access is very often not straightforward compared to regular volatile memory access. This work introduces a new programming model for persistent memory. The programmer can implement persistent memory operations with an existing library as backend. An additional layer allows an application to seamlessly adopt and interact with the backend for its persistent memory accesses in a similar fashion with the regular volatile heap memory accesses. The source code is available on GitHub [30].

Contents

1 Introduction

Conventional computer architectures usually differentiate between two data placement layers: volatile memory (e.g., DRAM) and persistent storage (e.g., SSDs). Programs use DRAM to store low latency volatile data that is required during runtime. On the other hand, disk based storage is used to store non volatile high volume data which is needed across program restarts. Persistent memory fills the gap between those two system stack layers by providing high capacity and low latency memory access which can also be non volatile. In the background section I will cover more details about this technology. [11]

A suitable use case for this new type of memory are key-value (KV) stores like Redis. Since volatile memory is expensive, it is most of the time a limited resource, which results in less stored data. Additionally due to the lack of persistence, the programs have to rely on different techniques to save the current memory data on disk based storage. Redis [32], for example, allows to take snapshots of the current data in specific intervals [33]. To this end, regular databases like PostgreSQL [31] can utilize persistent memory, since disk based programs usually have to use intelligent caching to reduce high latency memory access.

Programmers can interact with persistent memory via different libraries, whereby PMDK [28] is currently the most popular. Most libraries use different approaches for pointer representation, memory access and integration into the used programming language. This work highlights the following problems with the current approaches:

1. Persistent memory programming requires different data structures and a different data flow compared to regular DRAM applications.

2. It is harder to develop persistent memory based applications in the first place or to convert existing programs to support persistent memory, especially due to its crash consistency requirements.

3. The applications are heavily depending on one single library, which makes switching to other libraries harder and less comparable.

This work solves these problems by introducing a runtime library which acts as a layer above existing backend persistent memory libraries. The backend library can be swapped or adjusted without altering the actual user program. Additionally, we create

a compiler which transforms regular c style pointer access into function calls of the runtime library. Since virtual memory access should be still allowed the compiler also detects which pointers are virtual memory pointers and which pointers are persistent memory pointers by deduction at compile time. The compiler inserts function calls from the runtime library to correctly simulate native operations on persistent memory pointers.

We perform three types of evaluation: The performance benchmarks show that the runtime library results in a minimal performance overhead. The qualitative and quantitative evaluation of the compiler features clarify that the automatic type deduction and native pointer operations help to port existing C programs to support persistent memory, but also hit their limits. In the last evaluation we create an additional backend implementation and test which impact this has on the source code of a user program.

Overall this work has two main contributions:

1. Introduction of an abstraction of existing persistent memory approaches and programming models.
2. Provision of a persistent memory type deduction system, which analyses source code to identify, insert or modify persistent memory management operations.

In the following section we will cover the background details of persistent memory and will explain current approaches for this technology. Afterwards we introduce the design of our programming model and elaborate on the implementation details. At the end we will evaluate our model across multiple dimensions and will cover possible future concepts and implementations.

2 Background

2.1 LLVM

LLVM [36] is a toolchain which allows to build modular compilers. The fronted of the chain parses source code of a high level programming language and converts it into an intermediate representation (IR). The IR is platform independent and allows programmers to additionally analyse or run transformations over it. The backend transforms the IR into into a native assembly code for a specific target architecture. Assembling the generated source code creates multiple object files, which are linked in the end to a single one. [19, 1, 38]

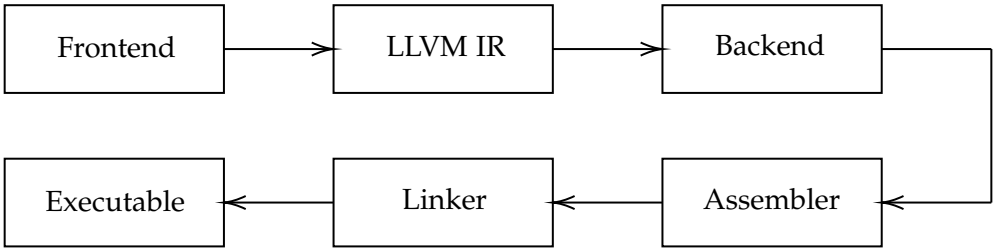


Figure 2.1: LLVM compailer chain which results in an executable

2.2 Clang

Clang [6] is a frontend for LLVM and allows to create targets for C, C++ and other programming languages of the C family. It takes as an input the source code and produces an abstract syntax tree (AST) out of it, which contains various information about the parsed program. Three different interfaces allow programmers to interact with the Clang toolchain:

1. LibClang - Provides a C interface to interact with Clang through other programming languages
2. Clang Plugins - Allow to inject actions into the frontend compilation process

3. LibTooling - Allows to create a standalone executable which performs operations on source files

Since LibTooling provides us enough freedom to run multiple compilations steps we choose this approach over LibClang and Clang Plugins. [5]

2.2.1 Abstract Syntax Tree

Each source file can be seen as an Abstract Syntax Tree (AST), which represents source code as relationship between nodes. These nodes contain function, variable or struct declarations but also statements like *if* and *while* statements. Expressions are also considered as statements in a Clang AST. *Decls* and *Stmts* stand for the respective base node classes in the Clang C++ library from which other statements or declarations are derived from. The AST also contains references between nodes in form of raw pointers, which allows simple navigation among them. An AST context holds additional data about node relations and source locations. Latter contains the specific start and end positions of a node in the source code file. [15]

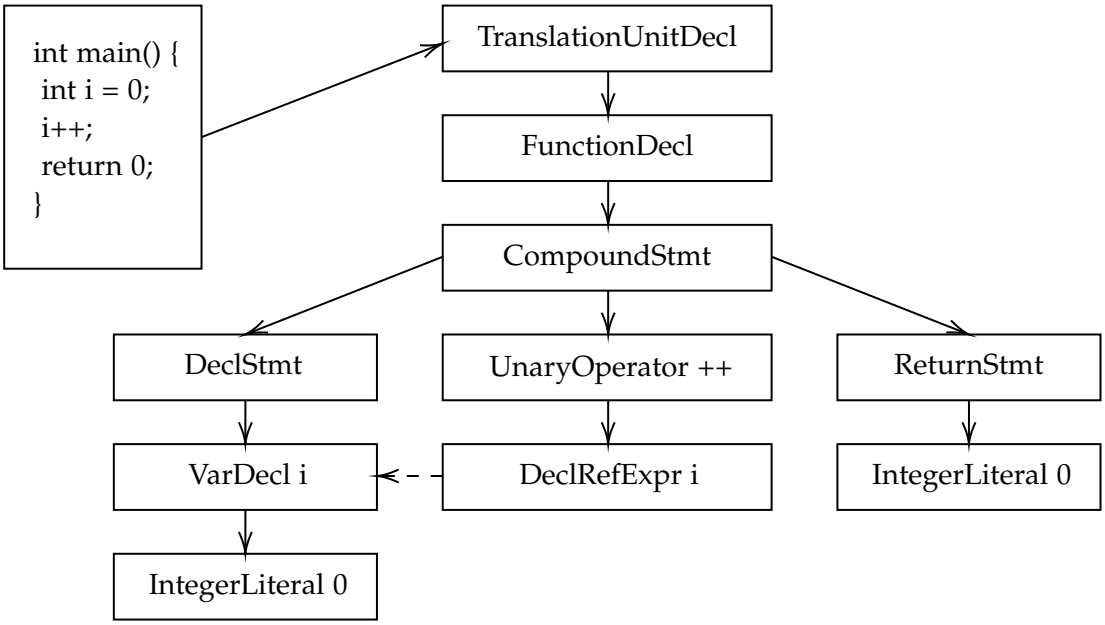


Figure 2.2: Source code as AST representation

Figure 2.2 illustrates an AST representation for a source file. The name of each node stands for the respective C++ class name. First the program declares a variable and initializes it, which is represented by the left branch of the AST. The middle branch

references the variable declaration node *VarDecl* with a pointer and applies a binary operator on it. In the left branch a integer literal is returned.

2.2.2 Traversing the AST

There are two built-in ways to traverse through an AST. The first is using *matchers*. it allows to specify requirements for a case to match on it. For each match a callback function will be invoked with the matched result as a passed parameter. On the other side it is possible to recursively traverse through an AST via a visitor pattern. This is helpful when analysing multiple different node types in a specific order is required. We will show this approach in more detail in the implementation part. [7, 20]

2.2.3 Rewriting source code

To modify the source code of the input files there are two theoretical ways. The first way would be to modify directly the objects of the AST. The class interfaces allow to create nodes and to modify member variables to some degree. In general this approach is not officially supported by Clang. This becomes apparent when performing complex modifications like inserting new statements. Another approach is to modify the actual source code which is stored in a string buffer. Clang allows this through a *Rewriter* [8] class. Since the source locations for every node in the AST are known, *Rewriter* enables us to insert text at these locations or to directly overwrite specific source location ranges.

2.3 Persistent Memory

Persistent memory usually refers to type of memory mainly produced by Intel as its Intel Optane persistent memory series. It is available as 128 GiB, 256 GiB and 512 GiB models and is directly connected to a DDR4 compatible RAM slot, which means its data is byte size addressable via the memory bus. This stands in contrast to NAND SSD connections which are realized with the PCIe interface and have to retrieve always a full block of data which can have a size up to 8kB. This enables persistent memory to have faster data access times which are close to DRAM, but still has the potential to provide a higher capacity than DRAM at a lower cost. This relationship can be visualized with a memory storage hierarchy. [12, 11, 29, 17]

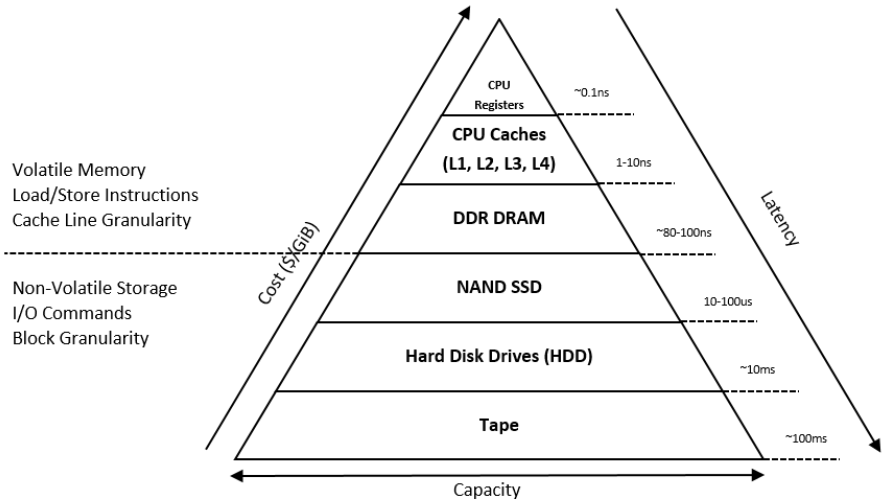


Figure 2.3: Storage hierarchy [29]

Figure 5.10 illustrates each type of storage, whereby capacity is represented by the horizontal axis and cost per byte together with latency represented by the vertical axis. Persistent memory creates a new option in this hierachy which is located between DRAM and NAND SSDs.

2.3.1 Programming Model

There is a standardized programming model for working with persistent memory created by the Storage Networking Industry Association (SNIA) [35]. It defines that the address space within the persistent memory is organized as a file system which is aware that it operates on a persistent memory device. The operating system uses

system calls to open and close these files and provide access rights which involves the kernel. On the other side reading data or writing data to a persistent memory file is realized with memory mapping. After opening a file the process maps the addresses of the persistent memory file to its virtual address space. On linux the system call *mmap* [24] is providing this functionality. This allows direct access (DAX) from the user space to the persistent memory data without involving additional system calls to the kernel. [34, 12]

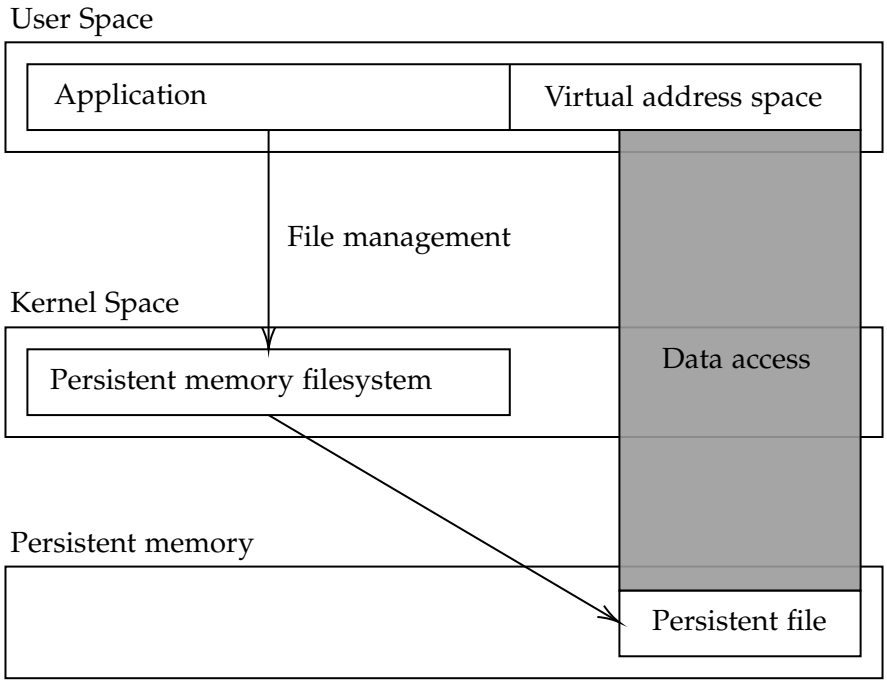


Figure 2.4: Persistent memory programming model

Figure 2.4 illustrates the connection between user space programs and persistent memory. The left side illustrates the file management task and the right side the memory mapping of the opened file.

2.3.2 PMDK

PMDK [28] is a collection of libraries which builds on top of the persistent memory programming model and supports the developer with multiple concepts that are introduced with this new type of memory. libpmem [13] provides low level functionalities such as persistent memory file mapping and flushing of data changes. libpmemobj [14] adds an additional layer on top of libpmem and introduces new techniques to handle

persistent memory data, which will be showcased in the following parts.

Persistent pointers

Every time a file is opened via PMDK the persistent memory addresses are mapped to virtual memory addresses. This means that programs can use this data similarly to usual heap allocated memory in DRAM. Data structures, for example a linked list, require often that virtual memory addresses are directly stored in the heap. The virtual address space of a program can differ each time the same program is run. For data stored in DRAM this does not represent a problem, since its data is expected to be volatile, which means that the data will lost after stop of the program. On the other side data stored in persistent memory is expected to still exist after a program restarts. This means that a virtual memory address which is stored in persistent memory can become invalid. Libpmemobj introduces persistent pointers, which represent a specific persistent memory address which are independent of virtual address mapping. [25]

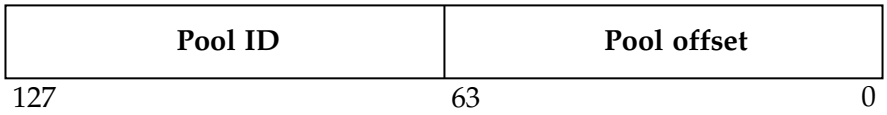


Figure 2.5: Memory layout for a persistent pointer

A *pool* represents a memory mapped file that has a unique id assigned to it. Offset stands for the address offset within the pool. The pool id enables to find the start location of the memory mapped file in the virtual address space. The virtual memory address of a persistent pointer can be then calculated by adding the offset to the pool address. The null pointer is represented by setting both values to zero. [25]

```
1 PMEMoid oid = {};
2 pmemobj_zalloc(pool, &oid, sizeof(int), ...);
3 int *ptr = (int *)pmemobj_direct(oid);
4 *ptr = 1;
5 pmemobj_persist(pool, ptr, sizeof(int));
```

Figure 2.6: Example for persistent pointer usage

Figure 2.6 shows how a persistent pointer, which is declared by the struct PMEMoid, can be used with libpmemobj. We allocate an integer in persistent memory and transform the resulting persistent pointer into a virtual memory pointer with the function pmemobj_direct. After this transformation the allocated integer can be used like a normal integer pointer. To make sure the applied changes have been flushed

we have to call pmemobj_persist. This example already shows that using persistent pointers requires a different programming style compared to normal pointers, since one has to perform pointer transformation by hand, passing around a pointer to the allocated pool and make sure that data is being flushed. Additionally PMEMoid has no type information about the data it is pointing to. PMDK provides a way to tackle pointer transformation and lose type information. One approach is to support the developer with macros, which allow persistent pointers to embed types and simplify transformation. [25]

```
1 TOID(int) oid = {};
2 pmemobj_zalloc(pool, &oid, sizeof(int), ...);
3 int *ptr = (int *)pmemobj_direct(oid);
4 *D_RW(data) = 1;
5 pmemobj_persist(pool, ptr, sizeof(int));
```

Figure 2.7: Persistent pointer usage with macros

Figure 2.7 shows how the example from Figure 2.6 can be simplified by using TOID, which is internally a union type containing a PMEMoid member and a integer pointer member. We can access the underlaying PMEMoid as integer pointer by using the D_RW macro, which will transform the persistent pointer into a virtual memory address by invoking pmemobj_direct. This approach already improves type saftey, but still requires manual pointer transformation. Additionally structs which contain persistent pointers can hardly be reused for DRAM usage due to the type differences of PMEMoid and an ordinary pointer. There is also a C++ wrapper for libpmemobj which solves some of the mentioned problems by using templates and operator overloading. The problem with this approach is that existing programs which are written in C have to be ported to C++ before the persistent memory migration can happen. [13, 27]

Transactions

Persistent memory data survives program crashes in contrast to DRAM data which will be lost. This means that after the restart of the program there has to be a state which is either valid or recoverable. For this purpose libpmemobj introduces transactions which store a copy of modified data in persistent memory. Subsequently the actual data will be modified and flushed. If a crash happens during this process transactions are able to reapply the old data. The library also includes transactional and atomic allocations which allow to initialize allocated data in a failure safe way. This approach also solves the manual flushing problem, but creates another issue for developers since they have to track modified data manually by function calls, which can lead to consistency mistakes. [26]

PMDK summary

Overall PMDK provides strong low level support for persistent memory, but lacks in providing a developer friendly frontend interface. This means that on one hand it is more complicated to write a new program for persistent memory, but on the other hand it also is more complicated to migrate existing code to support persistent memory due to the different programming models compared to DRAM programs.

2.3.3 Atlas

Atlas [4] is another persistent memory library which was created by Hewlett Packard. It includes a compiler plugin and a runtime library. Latter provides interfaces for opening, closing and memory mapping regions in persistent memory. Those regions provide an address space for data and have a unique id assigned to them similar to libpmemobj pools. Additionally to transactional interfaces, Atlas also puts strong emphasis on safe synchronisation locks across threads. To achieve this the compiler plugin inserts specific operations by applying a LLVM pass. In contrast to PMDK, Atlas provides no persistent pointer concept, instead the creators advise to make sure that the persistent memory is always mapped to the same virtual address space. [2]

```
1 int *ptr = (int *)nvm_alloc(sizeof(int), region_id);
2 *ptr = 1;
```

Figure 2.8: Allocation with Atlas runtime library

In Figure 2.8 we can see that the allocation of an integer and the subsequent operation on the resulting pointer is very similar to DRAM allocation and is in strong contrast to

Figure 2.7. Currently Altas is still in its experiential phase and there is no indication that HP is actively working on it. But the conflict between Atlas and PMDK shows that there are different approaches to persistent memory programming which can differ in implementation details and interfaces for the user. Apart from PMDK and Atlas there are also multiple other persistent memory libraries which have different focus areas, such as data analysis [16] or lightweight transactions [37]. If there are more popular persistent memory libraries in the future, programs heavily depend on one approach and require a lot of effort to be migrated to another library.

3 Related work

3.1 Transparent references

There are already approaches to use persistent memory transparently which means that pointers which point to persistent memory do not differ in syntax and semantics to regular pointers. One paper [39] proposes to use runtime checks which test if a pointer is a persistent pointer or a virtual memory pointer either on software level or with hardware support. If the pointer is a persistent pointer it will be transformed into its respective virtual memory address. There are three pointer types which can be detected.

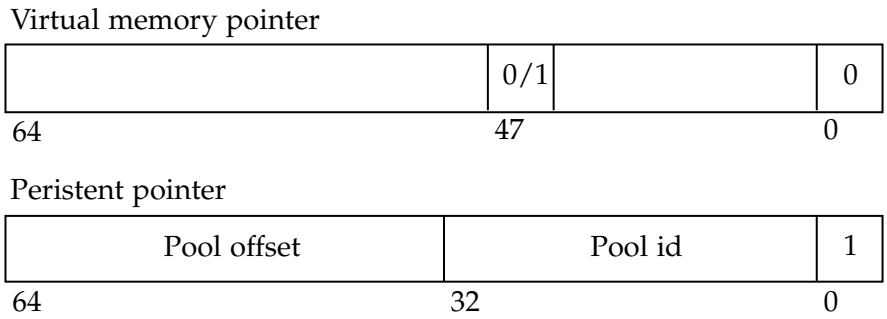


Figure 3.1: Pointer memory layout

The persistent pointer in Figure 3.1 stores both a pool offset and a pool id in a 31 bit memory layout similar to PMDK persistent pointers. The set bit indicates that the address is directly referencing persistent memory data. On the other hand the virtual memory pointer contains a 0 as its last bit and two possible values at bit 47 which either indicate that the address references to DRAM or to a memory mapped persistent memory address. With this differentiation a LLVM pass can insert operations to detect which type a pointer has at runtime. In case of a persistent pointer it will be translated into its respective memory mapped virtual address pointer. Additionally the software based checks can also detect some types at compile time by analyzing returned values from known functions which return persistent memory pointers, such as persistent memory allocation operations. But this inference already hits its limits

when passing a persistent pointer as function argument. In these cases dynamic checks have to be applied. The authors show that on average 42% of all pointer type checks have to be performed at runtime. The paper also demonstrates that detecting types on software level leads to performance losses caused by a higher ovearhead in general and branch mispredictions. Our work takes several contributions from this paper. We use a similar memory layout for persistent pointers but allow 64bit pool ids and pool offsets. Additionally we try to detect all pointer types across function calls at compile time.

3.2 Compiler support

There are also approaches to support persistent memory directly with a compiler for a specific programming language. Breeze [21] provides a C compiler that automatically inserts transaction logs for persistent memory pointers. It is required to use a seperate typing system for pointers and structs which point to NVM or will be stored in it. go-pmem [10] on the other hand is a fork of the Go compiler which allows transparent usage of persistent memory pointers. Both approaches do not support persistent pointers and use pointer recalculation techniques if the virtual address space location of a the memory mapped files change. We take inspiration from the overall system architecture from Breeze which includes the division between a runtime library and a compiler. At the same time we want to achieve a similar transparency level as go-pmem for the C programming language.

3.3 Meta libraries

We found no meta libaries for persistent memory which allow programmers to implement operations independently of the interfaces used in the source code. There is a project called memkind [3, 22] which is a meta allocator for different kinds of memory and built on top of jemalloc [18]. It is not able to support non volatile usage of persistent memory correctly since this would require additional interfaces, a generic meta allocator cannot provide. We take inspiration from this by creating a library which generalizes persistent memory behavour to allow different implementations with a single interface.

4 Design

Overall our design consists out of two parts: A frontend and a backend. The frontend is the actual source code a user writes. Our goal is that persistent memory interaction is as transparent as possible and close to working with volatile pointers, while still supporting specific characteristic of persistent memory explicitly. The backend consists out of user implemented persistent memory operations which are indirectly used by the frontend. This allows to change program behavior or to migrate backend libraries without modifying frontend code.

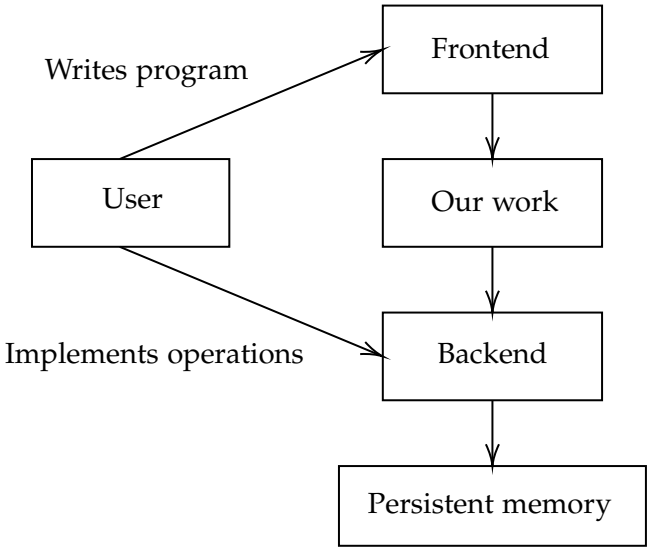


Figure 4.1: Proposed programming model

4.1 Frontend

This section covers the basic concepts the programmer faces, when developing a program with our compiler. In the implementation part we also explain some specific interfaces of the runtime library in more detail.

4.1.1 Initialization

At start of the program the runtime library has to be initialized with *pm_init*. The library uses a main region which is used for meta data but also for user data storage. During the initialization the developer can pass a structure which implements all requires persistent memory operations.

```
PmRegionConfig main_region_config = {
        .file_path = "./test",
        .root_size = sizeof(int)};

PmWrapperConfig config = {
    .backend = &PMDK_BACKEND,
    .main_region_config = main_region_config};

pm_init(config);
```

Figure 4.2: Runtime initialization

4.1.2 General pointer handling

The first step after the initialization is to get the root pointer of the main region. The root can store data on its own or point to other data structures. Its address is always retrievable by the runtime library which means it acts as an entry point for persistent memory data. pm_get_root function returns a new type of pointer, which is called

```
int *root = (int *)pm_get_root();
int value = *root;
```

Figure 4.3: Get the root pointer

PM pointer. PM pointers do not point to the virtual memory, but instead to a specific address in a specific region in persistent memory. On the other side there are VM pointers which point to a location in virtual memory. PM pointers allow exactly the same C syntax and semantics as VM pointers. The most common operation on pointers is dereferencing. In this case a regular C compiler would just emit an assembler instruction which loads the value at the location of the pointer address in virtual memory and stores that value on the stack, in a register or at another memory location. Since PM pointers do not represent a VM address, the attempt to load the value would either result in a segmentation fault or in reading an incorrect value. For that we

introduce a new compiler, built on top of Clang, which can detect PM pointers at compile time and will insert specific operations from the runtime library, depending on the operation performed on the pointer. The code insertions will be explained in more detail in the implementation section.

4.1.3 Pointer type deduction

The compiler can detect in many cases if a pointer is an address for virtual memory or for persistent memory at compile time. The starting point for this deduction are known functions from the runtime library which either expect or return PM pointers.

Function name	**Description**
pm_get_root	returns a PM pointer which indicates the start of the root of the region
pm_alloc and pm_calloc	returns a PM pointer which indicates the start of the allocated area
pm_free	takes a PM pointer to free the respective memory block

Table 4.1: Runtime functions which interact with PM pointers

All function variables, function return values and global variables which originate from external libraries are automatically declared as VM pointers if the respective type is a pointer. With this basis information the compiler can infer resulting pointer types

```
1 // Detected as VM pointer
2 int *ptr1 = (int *)malloc(sizeof(int));
3 // Detected as PM pointer
4 int *ptr2 = (int *)pm_alloc(sizeof(int));
```

Figure 4.4: VM and PM pointer differentiation

in complex programs by analyzing performed operations. In order to fully differentiate between different cases we introduce a new type system on top of the regular typing system, which we call "PMI" (Persistent Memory Indications). When we just use the term "type", we describe a regular C type.

PMI types PM and NULL PTR can be only assigned to expressions or variable declarations which evaluate to pointer types. All other PMI types can be assigned to

PMI type	Description
PM	Indicates that the expression or declaration is a PM pointer
NO PM	Indicates that the expression or declaration is not a PM pointer
UNINITIALIZED	Indicates that a variable declaration has not been initialized
NULL PTR	Indicates that a variables declaration was initialized with NULL
UNKNOWN	Indicates that the type of a expression or declaration is not detectable without additional information

Table 4.2: PMI types

any type. PMI types can be converted into other PMI types by specific operations. In Figure 4.5 addr is no PM pointer since it contains the address of ptr on the stack.

```
1 // PMI type PM
2 int *ptr = (int *)pm_alloc(sizeof(int));
3 // PMI type NO PM
4 int value = *ptr;
5 // PMI type NO PM
6 int **addr = &ptr;
```

Figure 4.5: PMI conversion examples

Therefore it is a VM pointer. The compiler evaluates for each expression the resulting PMI type depending on the PMI types inside the expression.

4.1.4 Function handling

Functions also can expect PM pointers as arguments and can return PM pointers. These PMI types are again deduced by the compiler.

In order to achieve PMI type deduction for functions the compiler has to start evaluating PMI types in the main function. From that point on he moves to every other function which was called inside the main function. The PMI types of the arguments for the function call are transferred to the parameter types of the called function. The compiler proceeds its evaluation in this function until the last return statement. The

```
1  // a: PMI type PM
2  // value: PMI type NO PM
3  // return value: PM type PM
4  int* set_value(int *a, int value) {
5    *a = value;
6    return a;
7  }
8  // PMI type PM
9  int *ptr1 = (int *)pm_alloc(sizeof(int));
10 // PMI type PM
11 int *ptr2 = set_value(ptr, 10);
```

Figure 4.6: Function handling example

resulting PMI type will be then transferred to the caller. In Figure 4.6 ptr1 must be PMI type PM. When the unknown function set_value is invoked the compiler remembers the types of the given arguments and jumps to the body of the called function. At the the end of the body of set_value the returned PMI type PM will be remembered and correctly assigned to ptr2. In case set_value would be called again, but now with a VM pointer, the compiler would return an error, since multiple PMI type signatures are not allowed for the same function.

```
1 // VM pointer
2 int* ptr3 = (int*)malloc(sizeof(int));
3 // Error: ptr3 is not a PM pointer
4 int *ptr4 = set_value(ptr3, 20);
```

Figure 4.7: set_value from Figure 4.6 is called with a VM pointer which results in an error

In case a function is never called with a PM pointer as argument or does not even take pointers as arguments, the compiler still has to evaluate that function, since it could call functions from the runtime library which would result in access to PM pointers or global state.

There are also limitations. A common practice is to not directly call a function, but instead store the address of the function in a variable to perform the call at a later stage. In this case the compiler will set the PM type of the variable as NO PM, since it would just hold a virtual memory address to that function. When performing the call operation on the function pointer, the compiler will not know to which function the variable is pointing to. In this case any parameter types are allowed.

Since the compiler has no indication with which argument types set_value will be

```
1  PM int* set_value(PM int *a, int value) {
2    *a = value;
3    return a;
4  }
5  // PMI type VM
6  int *ptr1 = (int *)pm_alloc(sizeof(int));
7  // PMI type NO PM
8  int* (*set_value_var)(int*, int) = set_value;
9  // PMI type PM by attribute
10 PM int *ptr2 = set_value_var(ptr1, 10);
```

Figure 4.8: Function pointer example

called, the programmer has to provide manual type attributes to all pointer types of the parameters. Those attributes are only allowed for function parameter declarations, function return type declarations and variable declarations. They will overwrite the detected pointer type from the compiler and can be only applied to pointer types. The attributes are provided as macros for convenience.

```
1 // PM pointer type
2 #define PM __attribute__((pointer_type(1)))
3
4 // VM/NO PM pointer type
5 #define VM __attribute__((pointer_type(2)))
```

Figure 4.9: PMI type attributes

Since the return value type of a invoked function pointer is also not known to the compiler, an attribute has to be also provided for the value which stores the result. If the function set_value was already directly called once before the indirect pointer usage, the compiler will know how treat this function and no explicit PMI types are required for its signature.

4.1.5 Array handling

Arrays in C are just pointers with a specific type. The address indicates the start of the array and the type the size of each element. They can be allocated and used with our compiler similar to regular C arrays.

In case the array is an array of pointers the compiler interprets each element as PMI type PM, since it does not make sense to store a VM pointer in persistent memory.

In case the array is of PMI type NO PM, a developer could store PMI type PM or NO

```
1  // Compiler knows type of a
2  int *set_value(int *a, int value) {
3    *a = value;
4    return a;
5  }
6
7  int *ptr1 = (int *)pm_alloc(sizeof(int));
8
9  // Direct call happens here
10 int *ptr3 = set_value(ptr1, 20);
11 int *(*set_value_var)(int *, int) = set_value;
12 // Call by pointer happens here
13 PM int *ptr3 = set_value_var(ptr1, 10);
```

Figure 4.10: Implict PMI type deduction for function pointers

```
1  int count = 10;
2  int *array = (int *)pm_alloc(sizeof(int) * count);
3
4  for (int i = 0; i <= 0; i++) {
5      array[i] = i * i;
6  }
```

Figure 4.11: Array handling

PM in it. The compiler will always assume that each pointer has PMI type NO PM and will print out a warning, which indicates that the programmer can use an attribute to correct the PMI type.

4.1.6 Struct handling

Structs and unions work similar to arrays. If the struct pointer has PMI type PM, each pointer within the structure is also a PM pointer. If the struct type is not a pointer or has PMI type NO PM, attributes have to be used to correct the PMI type of members.

4.1.7 Uninitialized variables

Uninitialized variables will be set to PMI type UNINITIALIZED. The first time a value is assigned to them, their type will be updated. It is not allowed to return uninitialized pointers or use them as function arguments.

```
1 int count = 10;
2 // PMI type PM
3 int **ptrs = (int **)pm_alloc(sizeof(int*) * count);
4 // PMI type PM
5 int *ptr = ptrs[0];
```

Figure 4.12: Array element PMI type deduction

```
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;
```

Figure 4.13: Struct handling

4.1.8 Null handling

PM pointers can be also set to NULL the same way as VM pointers. This means that error handling for pointers works like usual. If a pointer is initialized with NULL or is uninitialized and then updated to NULL, its PMI type will be NULL PTR. If the value of a pointer is set to NULL, which already has PMI type PM or NO PM no PMI type change will occur.

Functions can also return NULL values, but additionally have to return at least one pointer which is of PMI type PM or NO PM. Otherwise the return type has to be annotated with a PM attribute. It is not allowed to pass NULL PTR pointers as as parameters.

```
1 // type UNINITIALIZED
2 int *ptr;
3
4 // ptr is now type PM
5 ptr = (int *)pm_alloc(sizeof(int));
```

Figure 4.14: Uninitialized variable handling

```
1 // PMI type NULL PTR
2 int *ptr1 = NULL;
3 // PMI type PM
4 ptr1 = (int *)pm_alloc(sizeof(int));
5
6 // PMI type PM
7 int *ptr2 = ptr1;
8 // still type PM
9 ptr2 = NULL;
```

Figure 4.15: Null updating v

4.1.9 Multiple regions

In every example so far, we only accessed and allocated memory in the main region. It is also possible to operate on additional regions. One can open a region by invoking pm_init_reg which returns an unsigned 2 byte reference id which identifies the region across restarts. Id 0 is used for error indication, similar to a null pointer. The main region always has id 1. For most runtime functions there is a version which the suffix _reg, which allows to specify the region reference id in order to operate on that specific region. Pointer type deduction and pointer operations work the same as described the sections before.

```
1 PmRegionConfig second_region_config = {
2     .file_path = "./test_second",
3     .root_size = sizeof(int)};
4
5 pm_region_reference_id id = pm_init_reg(second_region_config);
6
7 int *ptr = (int *)pm_alloc_reg(sizeof(int), id);
8
9 *ptr = 1;
```

Figure 4.16: Multiple region handling

4.1.10 Developer support

Since there are edge cases and limitations with the new compiler, it is important for the developer that he gets enough information about PMI type issues during development. On the one hand this information will be transported via the Clang diagnostics, which emit errors and warnings to the terminal similar to the usual Clang compiler. Additionally it can help to be able to comprehend the type of each function signature or variable while writing the code. To achieve this the compiler supports integration into code editors and IDEs such as Visual Studio Code.

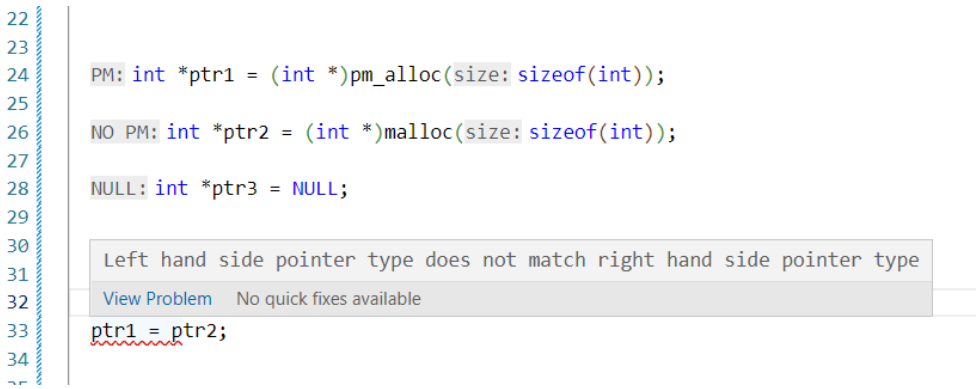


Figure 4.17: VSCode PMI type indications and error messages

4.2 Backend

This section will cover the concepts for backends. Those implement the basic operations on persistent memory which are directly invoked by the runtime. In order to create a backend, the programmer has to provide an implementation for a structure which store function pointers. This pattern is also called factory pattern. To most functions a PmBackendContext pointer will be passed as first argument. This context contains all necessary information for the currently accessed region.

Operation	**Description**
init	Allows to initialize used library
open_or_create	Create or openes a region
close	Closes a region
finalize	Allows to clean up used resources at end of the program
get_root	Gets the offset for the root
alloc	Returns the offset for the start of allocated peristent memory
calloc	Same as alloc but allocated memory is zeroed
free	Frees allocated memory
read_object	Transform the offset into a virtual memory address
write_object	Writes data to a given virtual memory address

Table 4.3: Backend operations which are required to be implemented

```
1 typedef struct PmBackendContext {
2   PmRegionConfig region_config;
3   pm_region_reference_id reference_id;
4   pm_region_id id;
5   void *data;
6 } PmBackendContext;
```

Figure 4.18: Backend context members

Figure 4.18 shows the structure of a backend context. reference_id represents the

16 bit region identifier for the frontend, while id represents a 64 bit region identifier for the backend implementation. The runtime is responsible for creating a mapping between reference_id and id. Additionally data allows to store custom data, which is required by some libraries such as PMDK.

4.2.1 Example implementation

This section states how to implement a backend with libpmemobj. We only show snippets of the actual code.

```
1  int open_or_create(PmBackendContext *context, bool *created_new) {
2
3    if (does_file_exist(...) == 0) {
4      context->data = pmemobj_open(...);
5      if (context->data == NULL) {
6        return 1;
7      }
8      *created_new = false;
9    } else {
10     context->data = pmemobj_create(...);
11     if (context->data == NULL) {
12       return 1;
13     }
14     *created_new = true;
15   }
16
17   context->id = pmemobj_root(...).pool_uuid_lo;
18
19   return 0;
20 }
```

Figure 4.19: open_or_create implementation with PMDK

In Figure 4.19 function open_or_create is being implemented which is one of the most important backend operations. It it initializes context members, which have to be present for other operations to work. We first check if the requested region file does already exist. If it exists we open the PMDK pool and store the result in the data member of the context. If the region file does not yet exist we create a new pool, We also store the new pool id int the context. This enables us now to implement additional operations such as the read_object function.

In Figure 4.20 we create a persistent pointer with the context member id as pool id and the passed function parameter offset as pool offset. pmemobj_direct_inline then transforms the persistent pointer into a virtual memory address. The implementation

```
void *read_object(PmBackendContext *context, pm_region_offset offset) {
  PMEMoid oid = {.pool_uuid_lo = context->id, .off = offset};
  return pmemobj_direct_inline(oid);
}
```

Figure 4.20: read_object implementation with PMDK

for write_object works similar to a typical memory copy function. In Figure 4.21 we delegate the input parameters to pmemobj_memcpy_persist which handles an optimized memory copy with additional flushing. In this case we also have to use the data member of the context to retrieve the pool address, since the function expects a PMEMobjpool pointer as its first argument.

```
void write_object(
    PmBackendContext *context,
    void *vm_ptr, char *data, size_t len) {
  pmemobj_memcpy_persist((PMEMobjpool *)context->data vm_ptr, data, len);
}
```

Figure 4.21: write_object implementation with PMDK

5 Implementation

5.1 Runtime

The runtime is implemented in C11 and responsible for calling and managing the backend implementation.

5.1.1 PM pointer representation

PM pointers in the frontend are just 64 bit values like usual virtual memory addresses. The upper 16 bit describe the region reference id and the lower 48 bit describe the offset within this region. The pointer can be represented in C as union type of a void pointer and a struct. This helps to convert between a pointer representation and a structured representation. The NULL pointer is represented by setting the offset and the reference id to 0. The main region always has reference id 1.

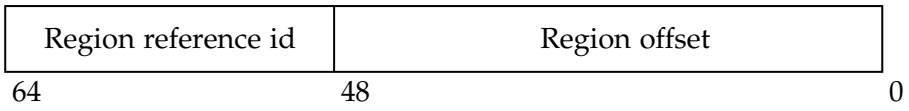


Figure 5.1: PM pointer memory layout

```
union PmThinPointer {
  struct {
    pm_region_offset offset : 48;
    pm_region_reference_id reference_id : 16;
  };
  void *ptr;
};
```

Figure 5.2: PM pointer C representation

5.1.2 Context handling

As already introduced each backend context represents an opened region with a region id, a region reference id and additional custom data. Every time a runtime function is called the respective backend context has to be found by region reference id and passed to the backend operation. To perform this task efficiently three steps are required during the initialization of the runtime library:

1. Try to open region with backend operation open_or_create
2. If the main region did not exist before, initialize a region id hash table in persistent memory
3. Allocate a backend context lookup table in DRAM

The region id hash table is a lookup table which maps region ids to region reference ids. It has an fixed entry count of 2^{16} for which every entry has a size of at least 10 bytes. It uses linear probing and a modulo based hash function. The data structure is allocated in the main region in persistent memory and will therefore exist across program restarts. It allows two operations: Function pm_region_reference_id will perform a lookup in

```
1 pm_region_reference_id rim_get_reference_id(pm_region_id region_id);
2 pm_region_reference_id rim_register_region(pm_region_id region_id);
```

Figure 5.3: Region id map operations

the hash table for the region id and returns the reference id. rim_register_region inserts an automatically generated reference id into the hash map, using the region id as key. The created value for reference id is always current entry count of the region id table plus 1.

On the other side the backend context lookup table maps region reference ids to backend context pointers. The structure will be stored in the heap and can be implemented in two ways, whereby both methods have drawbacks:

1. A fixed size array A of pointers which uses the reference id as index i. This requires to allocate $2^{16} * 8$ bytes of RAM from the beginning. The upside of this approach is that the lookup A[i] represents just a single memory lookup.
2. Use a dynamically sized hashmap. This has the upside that a smaller size of memory has to be allocated in total, since most of the time, only a small fraction of backend contexts are required at the same time in the memory. The downside is that the lookup algorithm will have an overhead, which was measured during performance evaluation.

In both implementations there are three operations provided, which act as their name suggests: The relationship between region id maps and context lookup tables can be

```
1 PmBackendContext *cm_get_context(pm_region_reference_id reference_id);
2 int cm_insert_context(
3     pm_region_reference_id reference_id,
4     PmBackendContext *context);
5 void cm_remove_context(pm_region_reference_id reference_id);
```

Figure 5.4: Context lookup table operations

also visualized like in Figure 5.5.

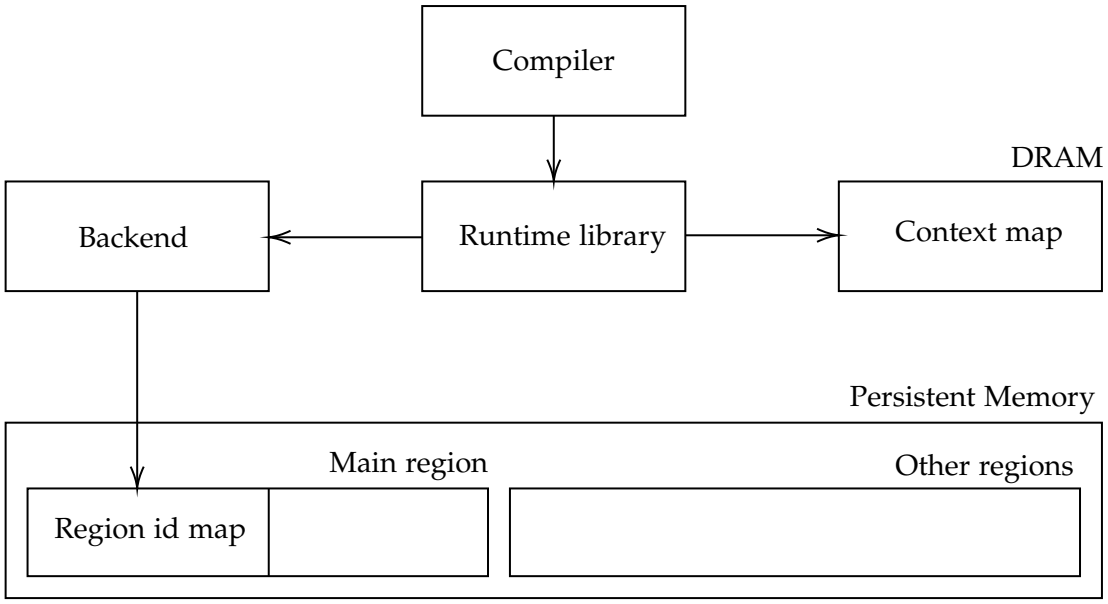


Figure 5.5: Overview of runtime components

Region initialization

The data structures we just introduced allow us now to manage more than just one region. When a programer calls pm_init_reg the following is happening:

1. A new backend context is being allocated
2. Call backend operation open_or_create

3. If the region did not exist before, register the region id in the region id hash table and get the new reference id
4. If the region was already created before, get the reference id from the region id hash table
5. Insert the created context into the backend context lookup table with the reference id as key
6. return the new reference id to the caller

Step 4 makes sure that a region id is always assigned to the same reference id across program restarts. On the other side the backend lookup table insertion in step 5 enables us to retrieve the created backend context from step 1.

Region operations

After initializing the regions, we can perform multiple operations with the returned reference id. In general every runtime function has a direct or an indirect parameter, which includes the reference id and therefore indicates on which region the operation has to be performed. The runtime function can then lookup the backend context by invoking cm_get_context. Afterwards the respective backend function can be called with the retrieved backend context as input. Examples for direct passing of reference ids are allocation functions. pm_alloc is just a shorthand function for calling pm_alloc_reg with region reference id 1. In Figure 5.6 we first retrieve the backend context by reference id. We then call the allocation function of the backend and return a PM pointer which consists out of the offset of the allocated region and the passed reference id.

```
1 void *pm_alloc_reg(int size, pm_region_reference_id reference_id) {
2   PmBackendContext *context = cm_get_context(reference_id);
3
4   pm_region_offset offset = config.backend->alloc(context, size);
5   return construct_pm_ptr(reference_id, offset);
6 }
```

Figure 5.6: Runtime alloc implementation

On the other hand indirect reference id passing happens via passing a PM pointer as parameter. The called function will extract the reference id from the passed pointer and then retrieves the backend context. pm_read_object and pm_write_object are the most important examples for this. In Figure 5.7 we first get the structured representation of

a PM pointer. Subsequently we also retrieve the backend context with the reference id from the passed PM pointer. In the end we call the backend operations for each runtime function.

```
1  void *pm_read_object(void *ptr) {
2    PmThinPointer thin_ptr = destruct_pm_ptr(ptr);
3    PmBackendContext *context = cm_get_context(thin_ptr.reference_id);
4
5    return config.backend->read_object(context, thin_ptr.offset);
6  };
7
8  void pm_write_object(void *pm_ptr, char *data, int size) {
9    PmThinPointer thin_ptr = destruct_pm_ptr(ptr);
10   PmBackendContext *context = cm_get_context(thin_ptr.reference_id);
11   config.backend->write_object(
12     context,
13     pm_read_object(pm_ptr),
14     data, size);
15 }
```

Figure 5.7: Read and write runtime implementation

5.2 Compiler

The compiler is built on top of LibTooling which contains multiple APIs to interact with the Clang compiler. It is written in C++. The compiler is an executable which takes in one C source file and outputs an object file which contains the compiled code of the input file. This object file can be then linked against the runtime library and the backend implementation. Figure 5.8 illustrates the full compiler chain which includes type evaluation and transformation of source code.

5.2.1 Type evaluation

With LibTooling, we can generate an AST from an input file. We can then perform PMI type evaluation on this AST that consists out of four components, whereby each component represents a C++ class. They are visualized in Figure 5.9.

VarManager

The main task of the VarManager is to store PMI types for variable and function declarations. An instance will be created at the begin of the type evaluation and

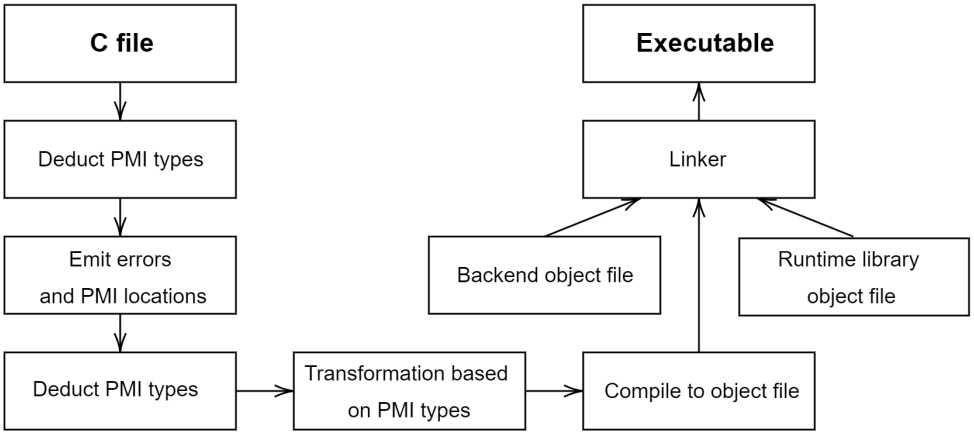


Figure 5.8: Compiler chain from C file as as input to executable

destroyed at the end of the full evaluation. It allows to get, set or check the existence of a variable and function PMI type by key. Those can be either a variable declaration or a function declaration address. Since those memory addresses are globally unique for the whole AST we do not have to differentiate between scopes of variable declarations. We store each entry in a map, using the declaration pointer as key and the PMI type as value. VarManager also contains utility functions for error and warning diagnostics and the functionality of printing all PMI type locations, which is be required for editor support.

GlobalEvaluator

The task of GlobalEvaluator is to prepare specific data which will be passed to FunctionEvalutor. This contains the following steps:

1. Finding the main function
2. Registering known function PMI types in the VarManager
3. Registering global variable declaration PMI types in the VarManager
4. Detecting functions which are never called directly
5. Running FunctionEvaluator for the main function

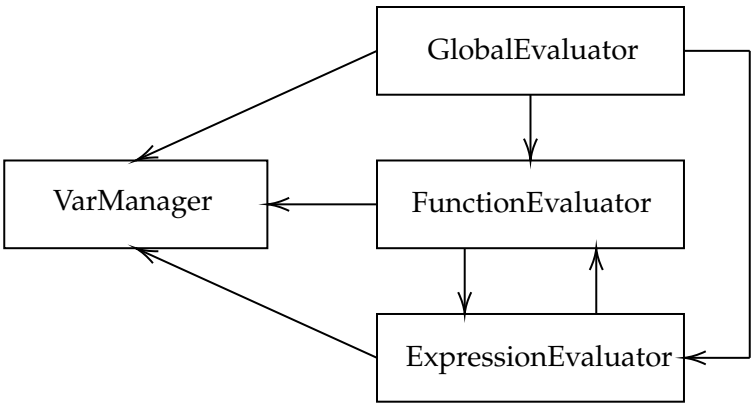


Figure 5.9: Type evaluation modules. Arrow from X to Y mean that X is depending on Y.

In GlobalEvaluator we have to traverse through the whole AST which means we can apply the RecursiveASTVisitor. GlobalEvaluator inherits from this class and can therefore override visitor functions. A specific visitor function will be called when the RecursiveASTVisitor visits a node with a specific statement or declaration type. When a visitor function returns false, the traversal will stop. The visitor functions in GlobalEvaluator always return true. In general we use a pattern to always return immediately when a specific condition matched, which improves readability in longer function bodies.

```
1  bool VisitT(T *stmtOrDecl) {
2
3    if (condition1) {
4      return true;
5    }
6
7    ...
8
9    if (conditionN) {
10     return true;
11   }
12
13   return true;
14 }
```

Figure 5.10: Visitor function pattern

First step is to find the main function of the program. We can achieve that by defining

a VisitFunctionDecl visitor function which can be seen in Figure 5.11. If a function declaration is a main function we store its pointer in a member variable.

```
bool GlobalEvaluator::VisitFunctionDecl(clang::FunctionDecl *fd) {

  if (fd->isMain() && fd->hasBody()) {
    this->mainFunction = fd;
    return true;
  }

  return true;
}
```

Figure 5.11: Finding the main function

It is important to note that the preprocessor was executed before the AST was constructed. This means that resulting declarations from header imports via the include directive will be present in the AST. We can use this fact for registering known function PMI types from the runtime library in the VarManager. All known function types will be stored in a map which has the function name as key and the function PMI types as value. The visitor function can then check if the function name of the incoming function declaration matches any predefined names. If there is a match, we can store its PMI type in the VarManager.

```
// Known function PMI types
pmWrapperKnownFunctionTypes = {
      {"pm_get_root", {
        .returnType = PointerType::PM,
        .parameterTypes = {}}},
      {"pm_alloc", {
        .returnType = PointerType::PM,
        .parameterTypes = {PointerType::NO_PM}}}
      //Additional runtime functions...
};

// Registering them in the VarManger in the visitor function
auto funcName = fd->getNameAsString();
if (isFunctionNameKnown(funcName)) {
    varManager.setFunctionType(fd, pmWrapperFunctionTypes[funcName]);
    return true;
}
```

Figure 5.12: Registering known functions

Other cases follow a similar pattern. For example for registering global variables we implement the function bool VisitVarDecl(clang::VarDecl *decl) and check via member function of decl if the variable is global. The variable could be unitialized or initialized with an expression. In the first case we directly store PMI type UNINITIALIZED in the VarManger. Otherwise we first have to evaluate the PMI type of the expression with an instance of ExpressionEvaluator. The last important step of GlobalEvaluator is to run FunctionEvaluator for the found main function.

FunctionEvaluator

FunctionEvaluator steps through each statement of a function body, evaluates for each of them the resulting PMI type via ExpressionsEvaluator and returns in the end the PMI type of the return value. This task can be again implemented via inheriting from RecursiveASTVisitor. In this case the tree traversal order inside functions is important, since subsequent PMI types of statements or expressions are depending on previous PMI types. RecursiveASTVisitor iterates through the tree using a depth-first algorithm.

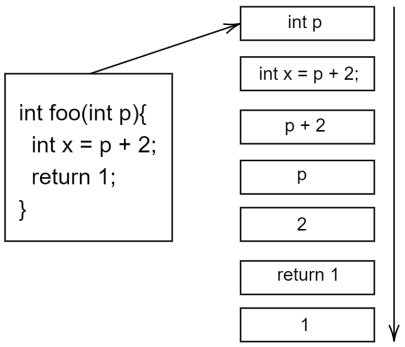


Figure 5.13: Depth-first evaluation order for program from top to bottom

FunctionEvaluator steps through each statement, which can contain a variable declaration. If the variable is directly initalized with an expression it will evaluate the PMI type of the expression with ExpressionEvaluator. The PMI type will be then stored again in the VarManager similar to global variables. We also have to evaluate all expressions which are not part of a variable declaration. These expressions could be part of an if condition, an assignment or a function call. For these cases we also use ExpressionEvaluator to evaluate subexpressions.

ExpressionEvaluator

ExpressionEvaluator evaluates expressions to a PMI type. In this case we will perform an inverse depth-first traversal.

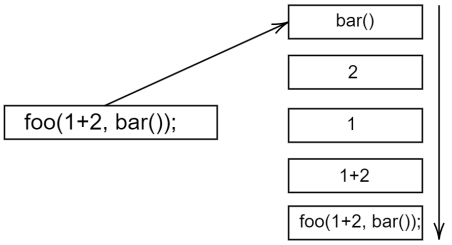


Figure 5.14: Inverse Depth-first evaluation order for expression from top to bottom

RecursiveASTVisitor does not support inverse depth-first traversal by default. A workaround is to first collect all subexpressions in depth first order with the visitor function VisitExpr in a stack. After the collection we can then invoke a second RecursiveASTVisitor for each item in the stack from top to bottom. This second instance will be the actual ExpressionEvaluator. For each visitor function of the second instance of RecursiveASTVisitor we have to return false so we do not traverse multiple times through the same expression.

Most of the work so far was only required preparation for the next steps. It is now possible to define for each expression type which PMI type it will result to, based on the PMI types of its subexpressions. In the end this will result in the final PMI type for an expression. In order to achive this correctly, ExpressionEvaluator has to implement a visitor function for every expression type which exists in C11. We use a map to store the PMI type of an expression by using the expression pointer as key. Other expressions can then retrieve the PMI type of the subexpression by performing a lookup in this map. We do not store those PMI types in the VarManager since they are not useful outside of the current expression.

Literal expressions can be seen as base case, since they do not depend on other factors and just result in a NO PM PMI type. Paren expression on the other hand just delegate the PMI type of its subexpression to the the top. Declaration reference expressions refer to a variable or a function. In case it is a variable we retrieve its PMI type from the VarManager since either GlobalEvaluator or FunctionEvaluator registered it there. We then store the PMI type of this expression in the internal map to make it available

```
bool ExpressionEvaluator::VisitIntegerLiteral(
    clang::IntegerLiteral *literal) {
  setType(literal, PointerType::NO_PM);
  return false;
}

bool ExpressionEvaluator::VisitParenExpr(clang::ParenExpr *expr) {
  setType(expr, getType(expr->getSubExpr()));
  return false;
}
```

Figure 5.15: PMI type definition for literal and paren type expression

for other expressions to lookup. When we visit a function declaration we first have to check if the function PMI type was already registered in the VarManager. If this is the case we set the type of the expression to the PMI type from the VarManager. Otherwise we have to evaluate this function with FunctionEvaluator.

```
bool ExpressionEvaluator::VisitDeclRefExpr(clang::DeclRefExpr *expr) {
  setType(expr, varManager.getVariableType(expr->getDecl()));
  return false;
}
```

Figure 5.16: Simplified declaration reference expression evaluation

5.2.2 Transformation

Since we evaluated the PMI type of every declaration, we can now transform the source code based on this information by inserting runtime library calls. As already explained in the background section we do not modify the AST directly. Instead we modify a buffer which contains the textual representation of the source code. A deserializer directly transforms an expression into a string representation of the transformed source. Rewriter allows us to replace the original expression in the source code with the transformed textual representation. This step will be performed directly after evaluating all PMI types of an expression. ExpressionEvaluator creates a new instance of ExpressionsWriter and passes the PMI types of all evaluated subexpressions to it. ExpressionsWriter also inherits from RecursiveASTVisitor and has to support every expression type which exists in C11. The traversal order is depth-first again.

Figure 5.18 showcases two simple examples for deserializing an expression to a string. An integer literal can be directly transformed into a string and written to string

```
bool ExpressionWriter::VisitIntegerLiteral(
    clang::IntegerLiteral *literal) {
  stream << exprToString(literal);
  return false;
}

bool ExpressionWriter::VisitParenExpr(clang::ParenExpr *expr) {
  stream << "(";
  TraverseStmt(expr->getSubExpr());
  stream << ")";
  return false;
}
```

Figure 5.17: Deserializing of expressions in ExpressionsWriter

stream which contains the full deserialized expression in the end. Parentheses on the other side just wrap braces around a subexpression which will traversed by calling TraverseStmt on it. Transformations have to be only performed for expressions types which have to behave differently depending on the PMI types of its subexpressions. The first expression we cover in more detail is the dereference operator on pointers. If dereferencing is performed on a subexpressions which has PMI type NO PM no transformation has to be performed. If the subexpression is PMI type PM a call to the function pm_read_object has to wrapped around this subexpression. A similar technique is used for member expressions on structs and subscript expressions on arrays.

```
// PMI type PM
PM int *ptr;

// Code written in frontend
int a = *ptr;

// Transformed code by the compiler
int a = *((int*)pm_read_object(ptr))
```

Figure 5.18: Transformed dereferencing behaviour

Assignment operators on PM pointers are more complex on the other hand. Consider the example in Figure 5.19. We could tranform the source code in the same way as in Figure 5.18. But we already covered in the background chapter that modifications to persistent memory should be logged or flushed. For this we allow the backend to implement a write operation to specify this behaviour. Our compiler has to insert these

write calls for some operators which also include assignments.

```
1 // PMI type PM
2 PM int *ptr;
3
4 *ptr += 1;
```

Figure 5.19: Transformed dereferencing behaviour

There are simple assignments which just assign one value to a variable, but there are also assignments which take the original value of a variable, perform a mathematical operation on it and store the result in the original variable. Additionally the left hand side will be returned by the expression. [23] We use a single pattern to support these cases. Figure 5.20 shows that we first create three variables. The first variable stores the PM pointer plus a possible offset, which can be used for structs or arrays. The second variable stores the value the PM pointer points to. The last operation saves the result of the assignment operation itself. We have to use these additional variables to avoid executing side effects which could be caused if the PM pointer expression is function that returns a PM pointer. The next step is to call the runtime function pm_write_object which indirectly calls the backend write operation. We concat all variables assignments, the function write function call and the return value of the assignment expression with the comma operator. This operator allows to combine multiple expressions to one expression which will be evaluated in order. The last expression in the chain is the resulting value.

Overall ExpressionWriter and ExpressionEvaluator cover a lot more additional cases such as the "address of" operator. The approach stays the same compared to the examples we discuss.

```
1  // PMI type PM
2  PM int *pointer;
3
4  // Code written in frontend
5  *pointer += 1;
6
7  // Transformed code by the compiler
8  int *ptr;
9  int ptr_value;
10 int assignment_value;
11 ((ptr = pointer + 0),
12  (ptr_value = *((int*)pm_read_object(ptr))),
13  (assignment_value = ptr_value + 1),
14  pm_write_object(ptr, (char *)(&assignment_value), sizeof(int)),
15  assignment_value);
```

Figure 5.20: Transformed assignment behaviour

6 Evaluation

We evaluate three dimensions for our work

1. We perform a quantitative evaluation on the the performance overhead.
2. We perform a qualitative and quantitative evaluation on migrating existing source code to our work
3. We perform a qualitative evaluation on implementing an additional backend.

6.1 Performance analysis

In this section we evaluate the performance of several programs which were implemented with our frontend. Since our work acts mostly as a wrapper we compare each program with a version that was directly implemented with PMDK. These versions use the approach shown in Figure 2.7, which means that a persistent pointer is first transformed into a virtual memory pointer before performing an operation on it. Subsequently the changed data will be flushed to guarantee persistence. If a part of persistent memory has to be copied pmemobj_memcpy_persist is used. To carry out correct comparison also the backend implementation for our work has to implement PMDK. On this side write operations are implemented with pmemobj_memcpy_persist and read operations with pmemobj_direct_inline. In both versions there are no transactions used. Table 6.1 shows a short description for each program we run our benchmarks for. Every program is compiled with -03 flags and Clang 13. We measure the execution time of each program with clock_gettime [9] and clock CLOCK_MONOTONIC and take the average of three runs to minimize occasional deviations. The host system is running a NixOS operating system with an Intel Xeon Gold 5317 as processor. It offers around 235 GiB of usable RAM and includes up to 996 GB of persistent memory which is provided via multiple Intel Optane Persistent Memory 200 Series sticks.

Figure 6.1 illustrates the runtime overhead for each program compared to raw PMDK. In all samples the PMDK version is faster. Program store and queue have the lowest overhead, since they always write to the same region. Our context map always caches the context for region reference id which was requested for the previous context lookup. This means there will be no additional context lookup required anymore. In contrast

Name	Description	Context lookup
store	Writes $50 * 10^7$ values to an integer array.	Fixed lookup table
multiregion 1	Inserts $25 * 10^7$ integers to 2 alternating regions.	Fixed lookup table
multiregion 2	Same as multiregion 1.	Hashtable
btree	Inserts 50000 key value pairs into a btree and read 50000 values.	Fixed lookup table
queue	Inserts $4 * 10^7$ integers and removes $4 * 10^7$ from a FIFO queue.	Fixed lookup table

Table 6.1: Benchmark programs implemented with our work and with PMDK

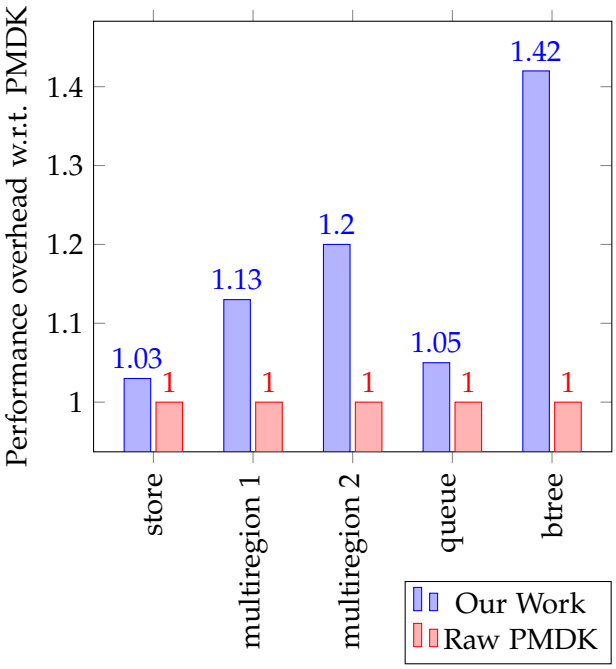


Figure 6.1: Benchmark results

multiregion 1 and 2 are not able to use this optimization method, since the program writes to two different regions in alternating order while the cache can only store one context reference. Multiregion 1 has less overhead than multiregion 2, which is caused by using a fixed size context lookup table over a hashtable. btree has the worst overhead. The reason for this is that the compiler inserts an object read function call in the condition for a while loop. In the PMDK version the call to pmemobj_direct_inline already happens outside of the while loop condition. Since it is possible to optimize this case manually with PMDK, the compiler could be improved to catch such cases and optimize them too. Overall the overhead does not negate the performance benefit persistent memory provides.

6.2 Program migration

Since our work has the goal to allow simple migration from existing DRAM applications to programs which support persistent memory we want to evaluate which code changes are required to reach that goal. We migrate a hashmap implementation which maps keys with arbitrary length to 64 bit values. The source code contains 12 functions and 128 arrow operators on structure pointers which can be potential persistent memory pointers.

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

Table 6.2: Required changes for hasmap migration

Table 6.2 shows all potential changes which have to be performed on the existing source code migration. There are no structure changes or pointer operation changes required which means our compiler is able to correctly detect and transform persistent memory pointers to virtual pointers. Only malloc, calloc and free functions have to be replaced with its respective persistent memory alternative from the runtime library. PMI type attributes are required on the one side for callback function parameters and

on the other side for out parameters from specific functions since our compiler does not offer support for this kind of parameter passing yet. The DRAM hashmap version stores only a pointer to the start of key which was allocated on the heap. In persistent memory this does not work since a lookup after program restart would cause that the key we want to compare with does not exist anymore. This means the hashmap has to manage a copy of the key in persistent memory. This requires to insert persistent memory allocations, performing the copy and freeing the key if it is not required anymore. This leads in total to 4 additional code blocks which modify the implementation. Overall our compiler helps to migrate pointer operations and structures to persistent memory with minor changes required. But there are also PMI types which are hard to detect for the compiler. Especially interacting with function pointers require frequent usage of attributes which can lead to manual errors caused by the developer. Source code which does not use functional pointer patterns work very well with our compiler. Improving the support for specific characteristics of persistent memory, which includes memory management of related resources, is not considered by our work.

6.3 Backend implementation

In order to test if our work correctly abstracts persistent memory behaviour and allows implementation with different libraries we additionally implement a backend with the library Atlas. Since we test most of our programs with a PMDK backend our goal is that no frontend code has to be changed in order to run every existing program with an Atlas backend. We had to perform minimal changes to the runtime library interfaces to support every operation correctly. For example Atlas requires to be initialized at the beginning and closed at the end independently of the region. For this we added an *init* and *finalize* operation to the backend interfaces. Apart from that no frontend code and also no logic in the runtime library had to be changed to support Atlas as backend. This means that complex programs can switch persistent memory libraries within a short time without migrating existing source code. The behavior of the programs itself is of course determined by the backend implementation. In case of Atlas this means that PM pointers do not contain an offset which is independent of the memory mapping, but instead a virtual memory address.

In Figure 6.2 we do not have to calculate the virtual memory address of an offset, since the offset itself is a virtual memory address. This means we just have to perform a cast to a pointer. Overall our work does not add additional features which did not exist before and acts only as a wrapper for existing libraries.

```
1 void *read_object(PmBackendContext *context, pm_region_offset offset)
2 {
3     return (void *)offset;
4 }
```

Figure 6.2: Backend implementation for a read_object operation with Altas

7 Future work

7.1 Transactions

Currently our design provides no interfaces for transactions. In the future backends could implement operations which implement transaction lifecycle events. As minimum one operation to start a transaction and one operation to stop a transaction are required. Additionally the backend has to implement either a logging operation for changes or log changes every time a write happens, which is showcased in Figure 7.1

```
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 }
```

Figure 7.1: PMDK implementation for transactional write

With transactional backend operations programs can explictly start and end transactions. Figure 7.2 modifies contiguous members of a struct within a transaction.

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

Figure 7.2: Explicit usage of transactions in the frontend

When using the approach from Figure 7.1 line 3 and 4 would both log the changes via pmemobj_tx_add_range even though one log would be sufficient. Our compiler could try to first detect cases in which transactions are required and secondly could try to use a single change log for multiple changes in contiguous memory.

7.2 Dynamic pointer evaluation

Our design still has problems to detect PMI types at compile time in several scenarios such as function pointers. One could combine compile time PMI type detection and dynamic pointer type checks at runtime. In contrast to the dynamic approach [39] we present in the related work chapter our work has the potential to detect most PMI types at compile time and only a fraction at runtime. When the PMI type of a pointer is UNKNOWN a specific bit in the pointer address can indicate if the PM pointer has to be transformed into a VM before before a specific operation will be applied.

```
1 UNKNOWN int *i;
2 return *i;
```

```
1 int *i;
2 if(is_pm_ptr(in)){
3     return *pm_read_object(i);
4 }else{
5     return *i;
6 }
```

Figure 7.3: Compiler inserts dynamic checks for specific operations

Figure 7.3 illustrates the compiler output for a dereferenced pointer with PMI type UNKNOWN. Function is_pm_ptr checks if the passed pointer is a PM pointer by testing if a specific bit is set. If the pointer is a PM pointer we have to apply a virtual address pointer transformation before the operation. One problem with this approach is that once a pointer is unknown, each other variable the pointer is assigned to, will be also UNKNOWN at compile time and would therefore require dynamic checks.

8 Conclusion

Overall we show that current persistent memory libraries differ in usage compared to DRAM programs but also in comparison among each other. For that we introduce a meta library which achieves semantics close to DRAM programs and allows developers to implement programs independent of the persistent memory library. We show that is possible to abstract persistent memory operations and that different pointer types can be detected at compile time with limitations. There is a only a small performance overhead when using an additional wrapper above existing libraries, which can be improved with further development. In some cases existing C code can be migrated to support persist memory with minimal effort, but in other cases some compiler hints have to be inserted. Overall our work provides the starting point for future improvements and implementations, such as automatic transaction insertion.

List of Figures

List of Tables

Bibliography

[1] *Assembling a Complete Toolchain*. URL: https://clang.llvm.org/docs/Toolchain.html. (accessed: 16.08.2022).

[2] *Atlas*. URL: https://github.com/HewlettPackard/Atlas. (accessed: 16.08.2022).

[3] C. Cantalupo, V. Venkatesan, J. Hammond, K. Czurlyo, and S. D. Hammond. "memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies." In: (Mar. 2015).

[4] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari. "Atlas: Leveraging Locks for Non-Volatile Memory Consistency." In: OOPSLA '14. Portland, Oregon, USA: Association for Computing Machinery, 2014. ISBN: 9781450325851. DOI: 10.1145/2660193.2660224.

[5] *Choosing the Right Interface for Your Application*. URL: https://clang.llvm.org/docs/Tooling.html. (accessed: 16.08.2022).

[6] *Clang: a C language family frontend for LLVM*. URL: https://clang.llvm.org/. (accessed: 16.08.2022).

[7] *clang::RecursiveASTVisitor< Derived > Class Template Reference*. URL: https://clang.llvm.org/doxygen/classclang_1_1RecursiveASTVisitor.html. (accessed: 16.08.2022).

[8] *clang::Rewriter Class Reference*. URL: https://clang.llvm.org/doxygen/classclang_1_1Rewriter.html. (accessed: 16.08.2022).

[9] $clock_gettime(3)$ *- Linux manual page*. URL: https://man7.org/linux/man-pages/man3/clock_gettime.3.html. (accessed: 16.08.2022).

[10] J. S. George, M. Verma, R. Venkatasubramanian, and P. Subrahmanyam. "go-pmem: Native Support for Programming Persistent Memory in Go." In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 859–872. ISBN: 978-1-939133-14-4.

[11] Intel. *Brief: Intel® Optane™ Persistent Memory*. URL: https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html. (accessed: 16.08.2022).

[12] Intel. *Introduction to Programming with Intel® Optane™ DC Persistent Memory*. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-programming-with-persistent-memory-from-intel.html. (accessed: 16.08.2022).

[13] Intel. *The libpmem library*. URL: https://pmem.io/pmdk/libpmem/. (accessed: 16.08.2022).

[14] Intel. *The libpmemobj library*. URL: https://pmem.io/pmdk/libpmemobj/. (accessed: 16.08.2022).

[15] *Introduction to the Clang AST*. URL: https://clang.llvm.org/docs/IntroductionToTheClangAST.html. (accessed: 16.08.2022).

[16] K. Iwabuchi, K. Youssef, K. Velusamy, M. B. Gokhale, and R. Pearce. "Metall: A Persistent Memory Allocator For Data-Centric Analytics." In: *CoRR* abs/2108.07223 (2021). arXiv: 2108.07223.

[17] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. Soh, Z. Wang, Y. Xu, S. Dulloor, J. Zhao, and S. Swanson. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module." In: Mar. 2019.

[18] *jmalloc*. URL: https://github.com/Aalto5G/jmalloc. (accessed: 16.08.2022).

[19] C. Lattner and V. Adve. "LLVM: a compilation framework for lifelong program analysis transformation." In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[20] *Matching the Clang AST*. URL: https://clang.llvm.org/docs/LibASTMatchers.html. (accessed: 16.08.2022).

[21] A. Memaripour and S. Swanson. "Breeze: User-Level Access to Non-Volatile Main Memories for Legacy Software." In: Oct. 2018, pp. 413–422. DOI: 10.1109/ICCD.2018.00069.

[22] *memkind*. URL: https://github.com/memkind/memkind. (accessed: 16.08.2022).

[23] Microsoft. *C Assignment Operators*. URL: https://docs.microsoft.com/en-us/cpp/c-language/c-assignment-operators?view=msvc-170. (accessed: 16.08.2022).

[24] *mmap(2) — Linux manual page*. URL: https://man7.org/linux/man-pages/man2/mmap.2.html. (accessed: 16.08.2022).

[25] Pbalcer. *AN INTRODUCTION TO PMEMOBJ (PART 1) - ACCESSING THE PERSISTENT MEMORY*. URL: https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-1-accessing-the-persistent-memory/. (accessed: 16.08.2022).

[26] Pbalcer. *AN INTRODUCTION TO PMEMOBJ (PART 2) - TRANSACTIONS*. URL: https://pmem.io/blog/2015/06/an-introduction-to-pmemobj-part-2-transactions/. (accessed: 16.08.2022).

[27] Pbalcer. *C++ BINDINGS FOR LIBPMEMOBJ (PART 1) - PMEM RESIDENT VARIABLES*. URL: https://pmem.io/blog/2016/01/c-bindings-for-libpmemobj-part-1-pmem-resident-variables/. (accessed: 16.08.2022).

[28] *Persistent Memory Development Kit*. URL: https://pmem.io/pmdk/. (accessed: 16.08.2022).

[29] *Persistent Memory Overview*. URL: https://docs.pmem.io/persistent-memory/getting-started-guide/introduction. (accessed: 16.08.2022).

[30] *pm-wrapper*. URL: https://github.com/Teppichseite/pm-wrapper. (accessed: 16.08.2022).

[31] *postgresql*. URL: https://www.postgresql.org/. (accessed: 16.08.2022).

[32] *redis*. URL: https://github.com/redis/redis. (accessed: 16.08.2022).

[33] *Redis persistence*. URL: https://redis.io/docs/manual/persistence/. (accessed: 16.08.2022).

[34] A. Rudoff. "Programming Models for Emerging Non-Volatile Memory Technologies." In: *login Usenix Mag.* 38 (2013).

[35] SNIA. *NVM Programming Model (NPM) Version 1.2*. URL https://www.snia.org/sites/default/files/technical-work/npm/release/SNIA-NVM-Programming-Model-v1.2.pdf, requested on 16.08.2022. 2017.

[36] *The LLVM Compiler Infrastructure*. URL: https://llvm.org/. (accessed: 16.08.2022).

[37] H. Volos, A. J. Tack, and M. M. Swift. "Mnemosyne: Lightweight Persistent Memory." In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XVI. Newport Beach, California, USA: Association for Computing Machinery, 2011, pp. 91–104. ISBN: 9781450302661. DOI: 10.1145/1950365.1950379.

[38] *Writing an LLVM Pass*. URL: https://llvm.org/docs/WritingAnLLVMPass.html. (accessed: 16.08.2022).

[39] C. Ye, Y. Xu, X. Shen, X. Liao, H. Jin, and Y. Solihin. "Supporting Legacy Libraries on Non-Volatile Memory: A User-Transparent Approach." In: *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. 2021, pp. 443–455. DOI: 10.1109/ISCA52012.2021.00042.