Persistent Memory Programming

Igor Chorążewicz

ABSTRACT

Persistent memory provides near DRAM latency, byte addressability, and data persistence. It delivers faster access to large datasets and quicker restart for many applications. Persistent memory can be seen as both storage and memory – it creates an entirely new tier in the storage hierarchy. This thesis describes a new programming model that allows developers to take full advantage of it. The main focus of this work is on designing efficient data structures. The first chapter states the objectives of the thesis and presents a brief history of persistent memory. The second chapter shows how the operating system exposes this new type of memory. It also describes challenges and programmers' responsibilities when using persistent memory. Chapter three presents the concept of a persistent programming language. Chapter four introduces several mechanisms for keeping data consistent, even in case of power failure or ungraceful system shutdown. It also discusses challenges with concurrent programming. The fifth chapter is an overview of existing persistent memory optimized data structures. It ends with several conclusions about designing data structures to provide the best performance. The sixth chapter is a detailed analysis of a concurrent radix tree included in the pmemkv database [19]. The described data structure can serve a general-purpose key-value store. Chapter seven focuses on methods of testing the correctness of persistent memory data structures.

Keywords: persistent memory, storage, data structures

Field of science and technology in accordance with OECD requirements: Natural sciences**,** Computer and information sciences

TABLE OF CONTENTS

ABSTRACT 3

1. INTRODUCTION AND OBJECTIVES OF THE THESIS 6

2. PERSISTENT MEMORY PROGRAMMING MODEL 8

2.1. SNIA programming models [51] 8

2.2. Platform and OS support 9

2.3. Persisting the data 11

2.3.1 Flushing 11

2.3.2. Non-temporal stores 12

2.4. Performance characteristics 13

3. PERSISTENT PROGRAMMING LANGUAGES 14

3.1. Object-oriented databases 14

3.2 Libpmemobj 14

3.2.1. Memory addressing 15

3.2.2. Data consistency and allocator 15

3.2.3. Type identification 15

4. DATA CONSISTENCY 16

4.1. Checking data consistency 16

4.1.1. Completed flag 16

4.1.2. Checksum and bit count 16

4.1.3. Failure atomicity within the same cache-line 17

4.2. Transactions 17

4.3. Copy on Write 20

4.4. Versioning 21

4.5. Concurrency considerations 22

5. RELATED WORK 25

5.1. Overview of design primitives 25

5.2 FPTree 26

5.3 BzTree 27

5.4. Write Optimized Radix Tree [9] 27

5.5. Pronto [18] 28

5.6. Summary 28
6. PERSISTENT RADIX TREE 29
6.1. Radix tree overview 29
6.2. pmem::obj::radix_tree 30
6.2.1. Leaf node layout 30
6.2.2 Internal node layout 30
6.2.3. Failure atomicity 32
6.2.4. Search operation 33
6.2.5. Insert method 34
6.2.6. Remove operation 37
6.3. Performance evaluation 38
6.4.1. Pmemkv persistent engines 38
6.4.2. Setup 39
6.4.3. Results 39
6.5. Summary 42
7. TESTING PERSISTENT MEMORY DATA STRUCTURES 43
7.1. Pmemcheck 43
7.2. Pmreroder 50
7.3. Summary 53
8. SUMMARY 54
LITERATURE 55

1. INTRODUCTION AND OBJECTIVES OF THE THESIS

The difference between access latency of primary and secondary memory in a traditional storage hierarchy is several orders of magnitude. Persistent Memory can fill this gap and increase most systems' performance. SNIA (Storage Networking Industry Association) defines Persistent Memory as "non-volatile, byte-addressable, low latency memory with densities greater than or equal to Dynamic Random Access Memory (DRAM)." [31]. The key aspects are persistence and low (new-DRAM latency). There are several solutions for achieving that. For example, the JEDEC standardization body defines three types of non-volatile dimms (NVDIMMs) [32]:

- NVDIMM-N – DRAM modules backed by batteries (or supercapacitors) integrated with NAND storage. If power is lost, an additional power source allows the DIMM controller to write DRAM content to NAND. On system restart, the controller would load the data back. One example of such NVDIMM is the AgigA Tech product [29].
- NVDIMM-F – flash memory that uses DDR bus
- NVDIMM-P – combines features of DRAM and flash

In 2019 Intel® released Optane™ DC Persistent Memory Module. It is based on Optane™ medium [50] and can retain data without any additional power source. This memory module also provides much higher capacities than traditional DRAM (128GB, 256GB, and 515GB capacities) [30], making it a good fit for handling large in-memory workloads. Figure 1 shows where non-volatile DIMMs fit in the storage hierarchy. One way of leveraging Persistent Memory along traditional storage would be to move latency-critical or frequently used data to PMEM. The performance gain would be high, especially for data accessed in a random pattern.

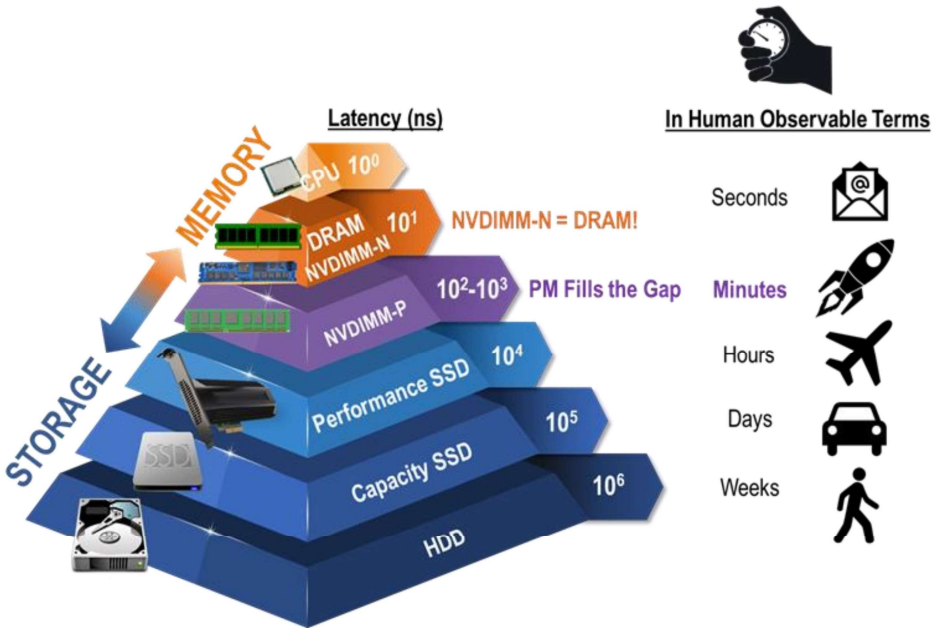

Figure 1 Storage hierarchy. Source: https://www.snia.org/education/what-is-persistent-memory

Persistent Memory has already seen adoption in the industry. One example is Apache Spark [26]. It uses Intel® Optane™ DC Persistent Memory Modules to scale up the computing servers with higher capacity memory. This limits the necessity to scale-out the computational instances and lowers cluster management and server maintenance costs. Another example is SAP HANA – an in-memory database management system, is another product that benefits from using Persistent Memory [27]. With DRAM only, SAP HANA loads all the necessary data from storage to main memory on startup. All operations on data are then performed in-memory. If SAP HANA instance must be restarted (e.g., in case of a failure), the application might experience reduced performance. With Persistent Memory, the initial load is no longer necessary. The data is retained across databases, and server restarts, reducing startup time.

Most of today's software assumes that the main memory is volatile. Just replacing DRAM with PMEM and expecting the data to be persistent will not work. Persistent Memory can be accessed using I/O routines developed for SSDs and HDDs, but this solution is viable only for large data transfers. In the case of small, byte-level accesses, the software would quickly become a bottleneck. To fully utilize Persistent Memory capabilities, a new programming model is needed. This thesis presents an overview of the model standardized by SNIA and describes challenges associated with it. I show that this model can be used to implement efficient data structures. Additionally, I also present how to test the correctness of Persistent Memory applications. To simplify code snippets in this thesis, most of the error handling and other miscellaneous C/C++ boilerplate will be omitted.

2. PERSISTENT MEMORY PROGRAMMING MODEL

This chapter presents a new programming model developed by SNIA. It shows how Persistent Memory is exposed to the programmers by the Operating Systems and what are the responsibilities of an application. In this work, Persistent Memory is mostly identified with Intel® Optane™ DC Persistent Memory Module. Some assumptions presented here (especially regarding performance) might not hold for future Persistent Memory devices.

2.1. SNIA programming models [51]

SNIA NVM Programming Model defines several programming modes for Non Volatile Memory (NVM):

- NVM.BLOCK
- NVM.FILE
- NVM.PM.VOLUME
- NVM.PM.FILE

NVM.BLOCK and NVM.FILE modes are used when NVM devices provide block storage behavior – they allow using NVDIMMs as SSDs. NVM.BLOCK is primarily used by file systems and other applications that are aware of hardware characteristics

NVM.FILE is a mode that does not require any knowledge of hardware. Existing applications that use the file I/O can work unmodified with this mode.

NVM.PM.VOLUME provides software abstraction for operating systems components. It gives a list of physical address ranges associated with PM volume and capability to determine whether PM errors have been reported.

NVM.PM.FILE provides user space applications a way to access NVM as memory directly. Although in this mode, Persistent Memory can be accessed using standard memory API (load/store instructions), applications have more responsibilities than when using DRAM. The most crucial aspect of Persistent Memory programming is taking care of data consistency. This will be described in detail in chapter three. Applications should also take into account differences in performance characteristics between PMEM and DRAM.

Figure 2 shows the different means of accessing NVDIMM.

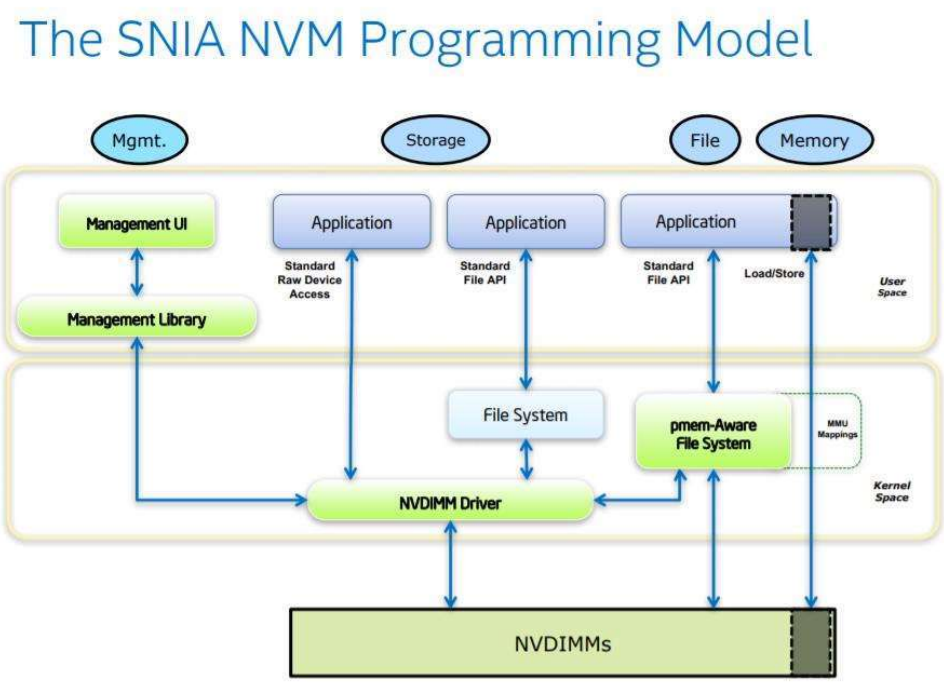

Figure 2 - SNIA Programming Model. Source: https://www.nextplatform.com/2018/02/07/momentum-gathers-persistent-memory-preppers/

2.2. Platform and OS support

In this thesis, I will focus on one of the previously mentioned programming modes – NVM.PM.FILE. This mode gives developers the most flexibility and control over how the data is transferred between CPU and NVDIMMs. This mode requires a Persistent Memory aware file system and exposes the Persistent Memory through memory-mapped files. Regular memory-mapped files allow accessing storage using load/store API with the help of a page cache. However, Persistent Memory is byte-addressable, and the cached pages would be unnecessary copies of the primary storage. That's why Persistent Memory aware file systems (for example, ext4 and NTFS) implement a DAX (Direct Access) feature, allowing memory mappings to access memory directly. On both Linux and Windows, Persistent Memory mappings can be created using native interfaces for mapping files (mmap() and MapViewOfFile(), respectively). Any writes issued to such memory mapping go directly to the DIMM bypassing the kernel.

When using Persistent Memory store/load API, it's crucial to define when a store is considered persistent. Figure 3 presents a power fail protected domain (a minimal and an extended version). Any store which enters this domain is considered persistent. The domain consists of a memory controller, and it's Write Pending Queue (WPQ), as well as the NVDIMM itself. If a power loss happens when the store is inside WPQ or is traveling to the NVDIMM, the platform must have enough stored energy for the store to complete their path to the medium. This

feature, flushing stores in case of power failure, is called ADR (Asynchronous DRAM Refresh) and is a platform requirement for supporting Persistent Memory.

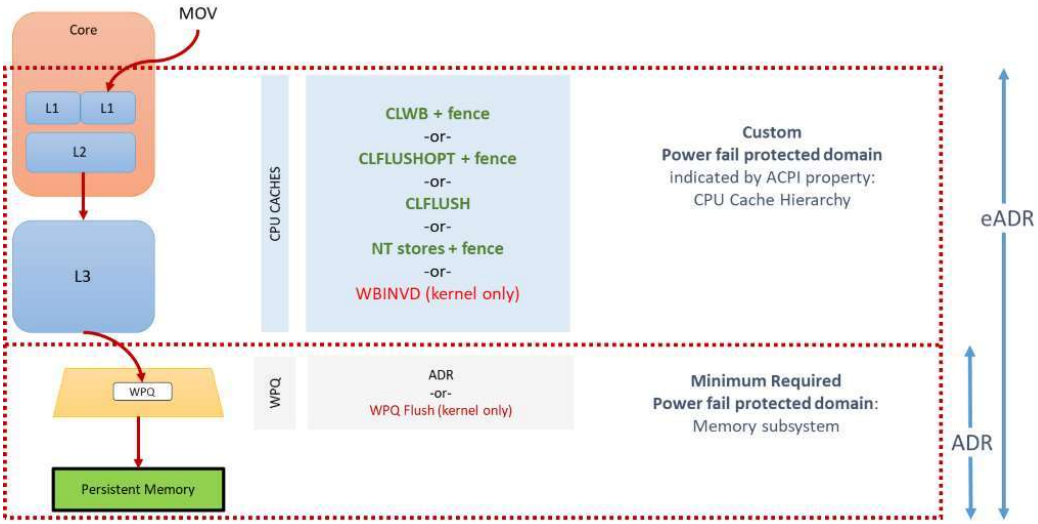

Figure 3 - CPU Cache hierarchy. Source: https://pmem.io/2019/12/19/performance.html

In general, it is an application responsibility to make sure a store has reached a power fail protected domain. This means that the application must either use non-temporal stores or flush the CPU caches. Non-temporal stores do not access the data in cache, new content is written directly to the memory. Flushing can be enforced by using *msync* syscall (or equivalent). However, a more optimal way is to use processor-specific instructions like CLWB, CFLUSHOPT, or CLFLUSH. They allow flushing a specific cache line, thus providing finer granularity than *msync* call, which operates on whole pages. Moreover, those instructions are available from userspace, which means there is no switch to kernel mode. The flushing instructions can only be used if a file is mapped with appropriate flags (e.g. MAP_SYNC flag is required on linux).

It's possible to design a custom power fail protected domain by also including CPU caches. In that case, a store is treated as persistent as soon as it enters the processor's cache. This feature is called eADR (Enhanced Asynchronous DRAM Refresh). Although eADR improves performance (some steps like flushing CPU caches are no longer needed), it introduces extra hardware maintenance costs.

Intel developed a Persistent Memory Development Kit (PMDK) to make persistent programming easier, architecture-agnostic, and OS-agnostic. The low-level library used in the following examples is libpmem [5]. It provides primitives for mapping files, flushing, and fencing. It also includes custom implementations of memcpy and memset optimized for Persistent Memory.

2.3. Persisting the data

All algorithms described in this thesis assume 8 bytes atomicity as guaranteed by x86. It means that if a power failure happens while performing an aligned 8-byte store, on the recovery, the destination memory region will either have all 8 bytes updated or all 8 bytes unmodified. When storing more than 8 bytes (or when a store is not 8 bytes aligned) to Persistent Memory, some additional techniques are needed to ensure data consistency. They will be described in the next chapter.

2.3.1 Flushing

To demonstrate the responsibilities of a program running on Persistent Memory, consider listing 1 (code is using libpmem[5] library). This simple program writes a string to pmem and uses a special flag to indicate completion. This flag can detect if the string was stored entirely or operation was interrupted somewhere in the middle. This detection is presented in listing 2

Listing 1 – Writing a string to Persistent Memory. Source: Own work.

```
struct data
{
    char string[4096];
    int completed;
};

int main()
{
    struct data *pmem = pmem_map_file("file", 4096,
                        PMEM_FILE_CREATE | PMEM_FILE_EXCL,
                        0666, NULL, NULL);

    strcpy(pmem->string, "Hello World!");
    pmem_flush(pmem->string, strlen(pmem->string) + 1);
    pmem_drain();

    pmem->completed = 1;
    pmem_flush(&pmem->completed, sizeof(pmem->completed));
    pmem_drain();

    pmem_unmap(pmem, 4096);
    exit(0);
}
```

Listing 2 – Detecting whether the string was written successfully. Source: Own work.

```
struct data
{
    char string[4092];
    int completed;
};

int main()
{
```

```
    struct data *pmem = pmem_map_file("file", 0, 0,
                        0666, NULL, NULL);

    if (!pmem->completed)
        exit(1); // string was not written entirely

    printf("%s\n", pmem->string);

    pmem_unmap(pmem, 4096);
    exit(0);
}
```

In code in listing 1, copying a string to Persistent Memory is followed by setting a *completed* flag. Between those two operations, there are two additional calls: *pmem_flush* and *pmem_drain*. *pmem_flush* call forces any changes in the specified range to be stored durably in Persistent Memory (it flushes the processor caches), and *pmem_drain* waits for any of those pmem stores to reach the persistent domain. In this example "completed" flag is needed to determine whether storing the "Hello World!" string succeeded. As seen in listing 2. *pmem_flush* operation is not transactional and can be interrupted by a power failure. The code assumes that the initial value of the *completed* flag is equal to 0.

Under the hood, *pmem_flush* calls architecture-specific processor instruction to flush CPU caches (or does nothing if eADR is detected). For Intel x86, this can be CLFLUSH, CLFLUSHOPT, or CLWB depending on the processor support. *pmem_drain*, on x86 architecture, is a memory barrier instruction – SFENCE. It enforces the stores' order (guarantees that pmem->string is stored before pmem->completed) and waits for all flush instructions to complete. It's also possible that *pmem_drain* is an empty function. That can happen when CLFLUSH is used in *pmem_flush* as CLFLUSH operations are ordered with respect to each other and with respect to writes, locked read-modify-write instructions, and fence instructions. All described instructions can be used at all privilege levels.

From a performance perspective, the preferred flushing instruction is CLWB. The reason is that unlike other flush instructions, it does not have to invalidate the cache line. This means that later reads to flushed location does not have to fetch that data from Persistent Memory.

2.3.2. Non-temporal stores

As mentioned earlier, instead of flushing CPU caches, it's possible to avoid using caches by employing non-temporal stores. Listing 3 presents a program similar to the one from listing 1. However, instead of *strcpy* and *pmem_flush* functions, *pmem_memcpy* is used. This function will either use regular stores followed by a flush or will use non-temporal stores. PMEM_F_MEM_NONTEMPORAL flag hints the library to prefer the latter. .

Listing 3 – Writing a string to Persistent Memory using non-temporal stores. Source: Own work.

```
struct data
{
    char string[4096];
    int completed;
};

int main()
{
    struct data *pmem = pmem_map_file("file", 4096,
                        PMEM_FILE_CREATE | PMEM_FILE_EXCL,
                        0666, NULL, NULL);

    const char* str = "Hello World!";

    pmem_memmcpy(pmem->string, str, strlen(str) + 1,
PMEM_F_MEM_NONTEMPORAL);

    pmem->completed = 1;
    pmem_flush(&pmem->completed, sizeof(pmem->completed));
    pmem_drain();

    pmem_unmap(pmem, 4096);
    exit(0);
}
```

2.4. Performance characteristics

To fully utilize Persistent Memory, it is important to know it's performance characteristics. In [3], researchers presented an overview of Intel® Optane™ DC Persistent Memory Module performance. The key observations from this paper are:

- Read latency of random memory load is 305ns (~3X higher than DRAM)
- Best store performance is achieved when using at least 256-byte blocks
- Reads scale with the number of threads
- Writes do not scale with the number of threads as well as for DRAM
- Write bandwidth is lower than read bandwidth (unlike for DRAM where bandwidth is symmetric)

3. PERSISTENT PROGRAMMING LANGUAGES

In the previous chapter's examples, a Persistent Memory pool was interpreted as a structure of trivial types (integers or strings). While this approach might be enough for the simplest programs, we need some additional abstraction to make Persistent Memory useful in building more complex applications. Specifically, a persistent programming language in which programmers could manipulate persistent objects as easily as volatile objects would be ideal.

3.1. Object-oriented databases

The concept of such a programming language is not new. In the late 1970s, when many applications started using relational databases, researchers came up with several prototypes of "persistent programming languages" [34]. Of course, Persistent Memory was not in the picture yet, but those languages were supposed to solve an "impedance mismatch" problem. The problem is as follows. Relational databases had their own naming system, their own data types, relations, etc. Any application that used such a database and wanted to process the data in the application code had to replicate all those things in its programming language. There was also a need for a conversion layer – application had to convert programming language objects to database objects and vice versa. One example of such an approach is Object-Relational Mapping. It Is implemented, e.g., by Java Hibernate [33]. The idea behind persistent programming languages was to integrate database management systems functionality more tightly into a programming language.

In the mid-1980s, several companies focused on persistent C++ [34]. In general, systems developed by those companies allowed to "persist" any C++ structures on disk. The main focus was on delivering high runtime performance, competitive with conventional C++. Several interesting Object-Oriented Data Bases (OODB) emerged. One example is ObjectStore [35]. It provides a way to manipulate persistent structures in a native C++ way and exposes transactions that allow programmers to save the changes durably on disk. The implementation takes advantage of memory mappings and page-fault handlers to load and cache data from disk. Unfortunately, the market for OODB never got very large. Most OODB vendors have failed or switched to offering some other products. However, I think that persistent languages and OODBs could be reborn with the emergence of Persistent Memory.

3.2 Libpmemobj

There is no modern programming language that would offer built-in support for persistent programming. However, some libraries allow programmers to store programming language objects (e.g., structs in C++) directly on PMEM. One such library is libpmemobj and libpmemobj-cpp [24] (a part of PMDK libraries). It provides memory pool abstractions (on top of memory-mapped files), a transactional object store, and a persistent allocator. Libpmemobj allows to relatively easy use Persistent Memory as a heap memory. A library such as libpmemobj has to solve several problems, which I will describe below.

3.2.1. Memory addressing

Because a memory pool can be mapped at different addresses in the process address space using volatile pointers, which point to a fixed address, is not possible. Data in a persistent heap is addressed using a special type of pointer – PMEMoid. It consists of a memory pool id and an offset within the pool. This architecture allows applications to have multiple pools open at the same time and to have cross-pool references. After the application restart, data inside the pool must be found somehow - there must be some top-level structure with a known address. Libpmemobj exposes a *root* structure to which all other objects can be attached. The *pmemobj_root* function can provide the pointer to this root object.

3.2.2. Data consistency and allocator

When modifying persistent objects, it is necessary to think about data consistency in case of power or application failure. Libpmemobj exposes transactions which allow grouping several operations into one atomic action. To ensure that the data is always in a valid state, a library can roll back or redo changes done in a transaction. Transactions, and other alternative mechanisms, will be described in more detail in the next chapter.

To allocate objects from a persistent heap, one needs a special allocator. It must perform all allocations atomically (with respect to power failure) and store it's metadata in PMEM. Libpmemobj exposes such allocator.

3.2.3. Type identification

A Persistent Memory pool can be accessed by multiple versions of an application (or even various applications). This means that those applications must use the same definition for all data types. Otherwise, if one application creates an object of type T and later, another application manipulates this object using a different definition, data might be corrupted. To avoid this problem, each object should contain enough information to identify it's type at runtime uniquely. Libpmemobj offers a partial solution – it provides a type number feature that allows associating allocated objects with an arbitrary number (which must be supplied by the programmers). This problem was also discussed in the paper describing the E programming language [37]. In the E programming language, type numbers were calculated as hash values of the class definition. Unfortunately, there is no way to generate stable type numbers automatically without compiler support in languages without robust support for runtime reflections (e.g., C or C++).

Another problem arises when a programmer wants to add a new feature that requires changing (or extending) the layout. To ensure compatibility between different versions of an application (or different applications), he must make sure that all those versions can handle both old and new data layout. This problem also arises in databases – it's referred to as schema evolution [38]. It's a hard problem, and solving it for Persistent Memory is also not easy. As a partial solution, one could use libpmemobj type numbers to detect different versions of the object and handle them in code differently.

4. DATA CONSISTENCY

This chapter describes techniques for ensuring data consistency. Firstly, I present methods for checking data consistency and integrity. Then, I show several mechanisms of making modifications larger than 8 bytes atomic.

4.1. Checking data consistency

Techniques presented here allow detecting if an operation succeeded without any interrupts. They form a building block for more advanced mechanisms like transactions.

4.1.1. Completed flag

The most straightforward mechanism for data consistency detection is storing a "completed" flag. This was already demonstrated in Listings 1,2 and 3. This approach can also be extended to support append-only logs. In that case, instead of a "completed" flag, the size variable should be stored. Writing to such log should be done in two steps:

1. Write new data at the end of a log and persist it
2. Increment size of the log and persist it

4.1.2. Checksum and bit count

Another, often more performant approach, is calculating a checksum of the data and writing this checksum alongside the data. The checksum approach is shown in listing 4.

Listing 4 – Using checksum to verify data consistency. Source: Own work.

```
struct data
{
    char string[4096];
    uint64_t checksum;
    char padding[64 - sizeof(checksum)];
};

int main()
{
    struct data *pmem = pmem_map_file("file", 4096,
                            PMEM_FILE_CREATE | PMEM_FILE_EXCL,
                            0666, NULL, NULL);

    struct data vdata;
    strcpy(vdata.string, some_long_string);
    vdata.checksum = checksum(&vdata.string, sizeof(vdata.string));

    pmem_memcpy(pmem-
>data, &vdata, sizeof(vdata), PMEM_F_PMEM_NONTEMPORAL);
}
```

To verify the data's correctness, it's enough to calculate a checksum and compare it to the checksum stored in Persistent Memory. If any event interrupts the write operation (and data has

not been entirely written), then the checksums will not match. Using checksums is often faster than a *completed* flag because it requires only one persist operation instead of two.

In listing 4, structure *data* is padded to a cache line boundary to optimize performance. Storing 64 bytes that fit within a cache using non-temporal stores can perform better than storing partial cache-line as a read-modify-write cycle is not required, and 64 bytes can be written using a single memory bus transaction [7].

Alexander van Renen et al. presented a very similar approach in [6]. The idea is to store a bit count of the data instead of a checksum. Although potentially faster than calculating checksum, this approach requires the memory to be initialized to zero before making modifications.

4.1.3. Failure atomicity within the same cache-line

Listing 5 shows an optimized way of ensuring data consistency for one cache-line. This example is very similar to the one in 2.1.1. The difference is that here only one cache line is modified. As a result, putting *pmem_drain* (a barrier) before setting a *persistent* flag is not necessary. All stores to the same cache line are ordered with respect to each other and with respect to flushing instructions.

Listing 5 – Failure atomicity within the same cache-line. Source: Own work.

```
struct data
{
    char A[32];
    char B[24];
    uint64_t persistent;
};

int main()
{
    struct data *pmem = pmem_map_file("file", 4096,
                            PMEM_FILE_CREATE | PMEM_FILE_EXCL,
                            0666, NULL, NULL);

    pmem_memset(pmem->A, 1, sizeof(pmem->A), PMEM_F_PMEM_NODRAIN);
    pmem_memset(pmem->B, 2, sizeof(pmem->B), PMEM_F_PMEM_NODRAIN);

    uint64_t persistent = true;
    pmem->persistent = true;
    pmem_flush(&pmem->persistent, sizeof(pmem->persistent));
    pmem_drain();
}
```

4.2. Transactions

A transaction allows grouping several operations (like setting a value or allocating memory) into one atomic action. Besides atomicity, transactions should also provide consistency, isolation, and durability. The set of these four properties is known as ACID. For Persistent Memory, atomicity is often guaranteed by using:

- Undo logs. When a memory region is about to be modified, the old value is read and stored in an undo log. After that, the snapshotted memory can be safely modified in-place. After all modifications are completed and persisted, the transaction completes and discards the log. If the transaction was interrupted (for example, by a power failure), data could be reverted to a consistent state by undoing all the changes. The undo log approach is presented in figure 4.
- Redo logs. This technique is also called write-ahead logging. Modifications are not done directly but are first recorded in the log. After the log is stored durably in Persistent Memory and marked as completed, all changes are applied. Unlike undo logs, old data does not have to be read, which can improve performance. One implication of using redo logs is that data is not immediately visible. The redo log approach is shown in figure 5.

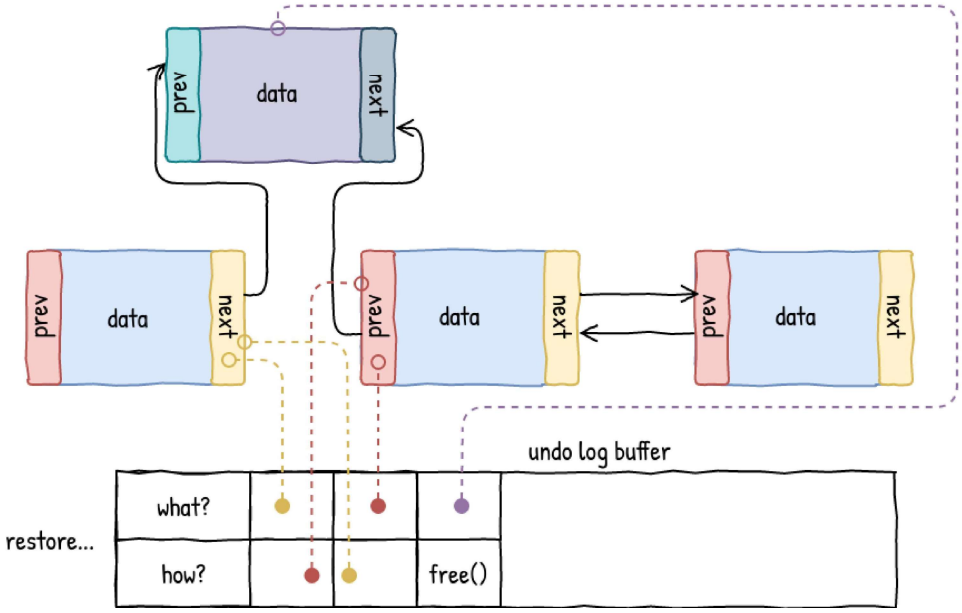

Figure 4 - Undo log. Source: https://pmem.io/2020/03/26/performance-2.html

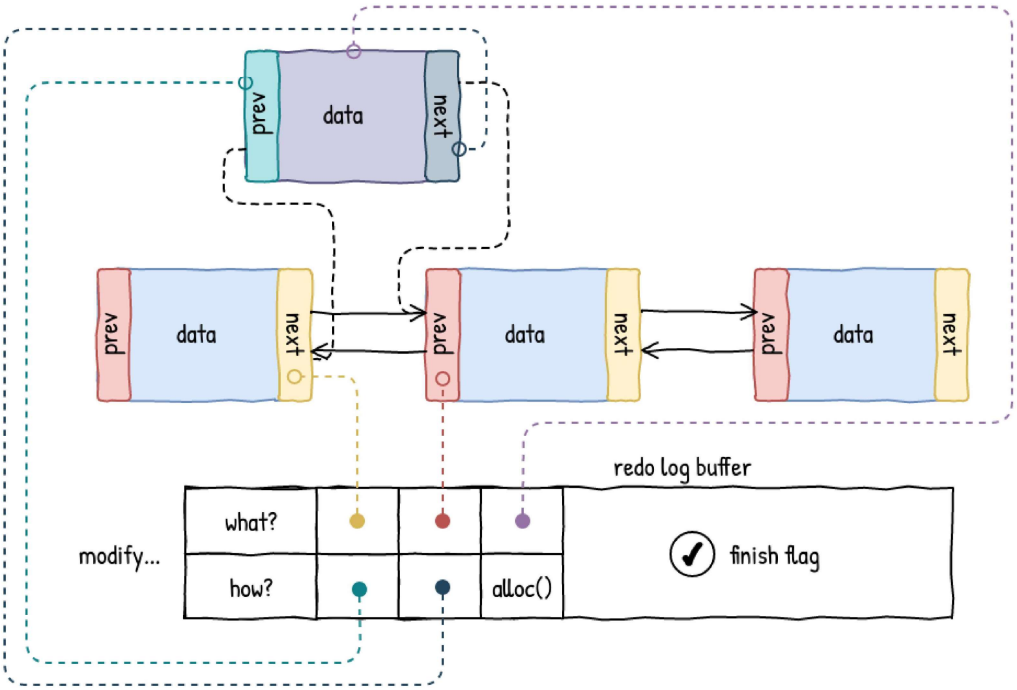

Figure 5 - Redo log. Source: https://pmem.io/2020/03/26/performance-2.html

To achieve durability, transactions must ensure that all modifications are flushed to Persistent Memory on commit. It is also essential to make sure that logs are stored durably before the data is modified.

Isolation can be achieved by taking a lock for the duration of the transaction. This way, the effects of transactions executed concurrently will be the same as if they were executed sequentially.

Using any logging technique introduces a noticeable performance and memory overhead. However, using special techniques, such as data-oriented design [22], can help significantly reduce that overhead. For example, if specific fields are often modified together (in a single transaction), then they should be stored next to each other. That way, there will be fewer (but bigger) snapshots, and the cost of persisting them will be lower [1].

Listing 6 presents an example code that uses pmem::obj::transaction.

Listing 6 – Example of pmem::obj::transaction. Source: own work.

```
struct root {
    int a;
    int b;
    pmem::obj::persistent_ptr<Object> c;
};

pmem::obj::persistent_ptr<root> r = pop.root(); // obtain pointer to a ro
ot object
```

```
// This makes 2 modifications and one allocation a single failure-
atomic action
pmem::obj::transaction::run(pop, [&]{
    pmem::obj::transaction::snapshot(&r->a); // add r->a to undo log
    pmem::obj::transaction::snapshot(&r->b); // add r->a to undo log

    r->a = 1;
    r->b = 2;
    // r->c is snapshotted automatically (by persistent_ptr operator=)
    r->c = pmem::obj::make_persistent<Object>();
});
```

4.3. Copy on Write

Instead of using redo or undo logs to record changes to a data structure, one could also use the Copy on Write (CoW) technique. The main idea is not to modify the data directly. The algorithm consists of three main steps:

1. Copy the original data to a new location
2. Modify the copy (modification does not have to be atomic)
3. Replace (a pointer to) old data with the updated data

The CoW is a generic technique and can be used for most data structures.

Consider a linked list *update* method. One possible implementation is presented in listing 7.

Listing 7 – Copy-on-write approach to updating node in a linked list. Source: Own work.

```
struct list_node
{
    char data[MAX_SIZE];
    size_t size;

    persistent_ptr<list_node> next;
};

void update(pmem::obj::pool_base &pop, list_node* prev, const char *data,
 size_t size)
{
    assert(prev->next);

    pmem::obj::transaction::run(pop, [&]{
        auto new_node = pmem::obj::make_persistent<list_node>();

        memcpy(new_node->data, data, size);
        new_node->size = size;
        new_node->next = prev->next->next;

        pmem::obj::delete_persistent<list_node>(prev->next);

        prev->next = new_node;
    });
}
```

For simplicity, the implementation in listing seven does not handle updating the head of the list. The update method still uses transactions for allocating and deallocation memory and performs snapshots of the next pointer but does not snapshot the data itself. Without transactions, one would need to implement a custom recovery procedure to avoid Persistent Memory leaks. For example, a pointer to the newly allocated node would have to be stored durably in Persistent Memory to either complete the update or free this node on failure.

Using CoW can limit the number of snapshots in the application. However, this comes at a cost – more allocations and deallocations. Since Persistent Memory allocator incurs significant performance cost, some researchers [13] concluded that CoW is not a good fit for Persistent Memory.

4.4. Versioning

Versioning is a concept similar to CoW, but instead of replacing the old data, multiple copies are accessible to users. Additionally, there is a *version* number that specifies the most current version of the data. Naturally, the memory overhead of versioning is higher than in CoW. However, the number of allocations and deallocations is much lower, which usually means better performance.

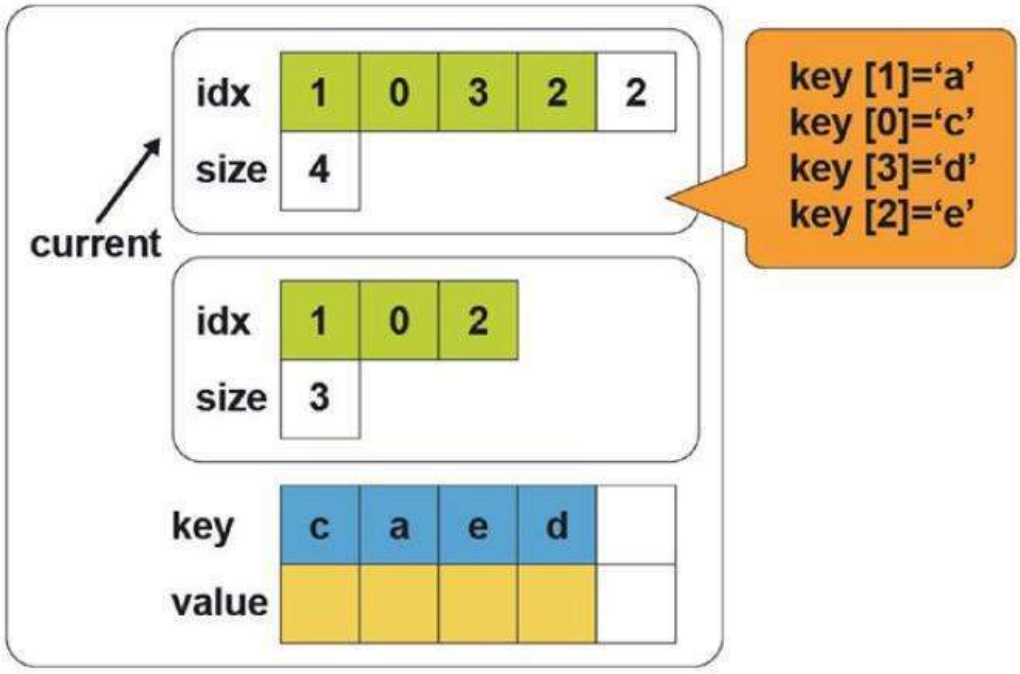

Figure 6 - Versioning. Source: Steve Scargall, A Comprehensive Guide for Developers, Chapter 11 [1]

The structure in figure 6 consists of 3 arrays and a *curr*ent pointer. The bottom array is an unordered array of key-value pairs. Two remaining arrays (index arrays) hold bottom array indexes in the order specified by the keys. One of them (the top one on the figure) holds the

current state (indicated by the *current* field). The second one (the middle one on the figure) is a *working copy* – it was valid at some point in the past.

To insert new key-value pair to such array and keep the ordering following steps are required:

1. Insert new key and value at the end of the bottom array
2. Copy the content of the current index array to the *working copy*
3. Insert index of new key-value pair at the appropriate place in the *working copy* and increase its size
4. Persist all changes
5. Change the *current* variable so that the working copy is now the current version.
6. Persist the *current* variable

Figure 7 presents the array state after insertion of key "b."

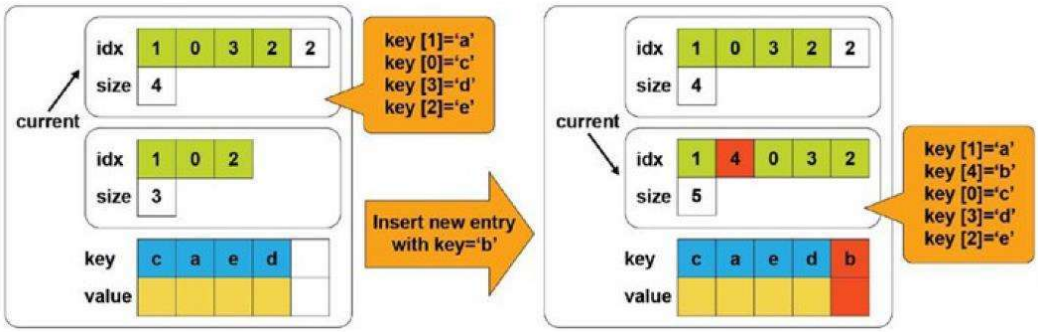

Figure 7 - Insert operation. Source: Steve Scargall, A Comprehensive Guide for Developers, Chapter 11 [1]

The state of the *working copy* array and the last slot of the key-value array does not have to be consistent during steps 1-4. Persist in step 4 ensures that all changes are stored durably before the *current* variable is modified. This guarantees that *current* variable always points to a data that is consistent and stored durably on PMEM.

4.5. Concurrency considerations

Ensuring data consistency is a challenging task. It gets even more complicated if data is accessed concurrently in a lock-free manner. A fundamental issue (if eADR is not used) is data visibility. When a thread issues a temporal (e.g., MOV) store instruction, the modification is visible to other threads before it is persistent (data can still be in cache). Consider the following scenario with one thread writing to a variable using atomic operations and the second one reading from it (listing 8 and 9):

Listing 8 – Atomic stores. Source: own work.

```
// thread 1
// initial value of pmem->a is 0
atomic_store(&pmem->a, 1);                 // visible = 1, persistent = ?
pmem_persist(&pmem->a, sizeof(pmem->a));   // visible = 1, persistent = 1
```

Listing 9 – Action based on a non-persistent variable. Source: own work.

```
// thread 2
// initial value of pmem->b is 0
if (atomic_load(&pmem->a) == 1) {
    pmem->b = 1;                           // visible = 1, persistent = ?
    pmem_persist(&pmem->b, sizeof(pmem->b)); // visible = 1, persistent = 1
}
```

Let's analyze the values of *pmem->a* and *pmem->b* from a second thread perspective. If a crash happened during the application execution, the possible values on restart are:

- (pmem->a = 0, pmem->b = 0) – e.g., when the crash happened before the first thread started
- (pmem->a = 1, pmem->b = 1) – e.g., when the crash happened after both threads completed
- (pmem->a = 1, pmem->b = 0) – e.g., when the crash happened after the first thread completed but before the second one started

But what happens if there was a crash just after the first thread set the value, but before it called *pmem_persist*? The second thread could have observed the new value (which is not yet persistent) and set the *pmem->b* variable to 1. On restart, it's possible that *pmem->a* is equal to 0, and *pmem->b* is equal to 1. All four combinations of values are possible on restart.

The fact that the second thread might take some actions based on a non-persistent state is problematic as it causes data inconsistency. One possible way of preventing this situation is by persisting all values just after reading them (but before any other action is taken based on the value). This guarantees that the value was persistent at some point in time. The second thread from the earlier example could look like presented in listing 10.

Listing 10 – Action based on the persistent variable. Source: own work.

```
// thread 2
// initial value of pmem->b is 0
if (atomic_load(&pmem->a) == 1) {
    pmem_persist(&pmem->a, sizeof(pmem->a));  // visible = 1, persistent = 1
    pmem->b = 1;                              // visible = 1, persistent = ?
    pmem_persist(&pmem->b, sizeof(pmem->b));  // visible = 1, persistent = 1
}
```

This solution makes (*pmem->a* is = 0, *pmem->b* = 1) case impossible. However, it introduces significant performance overhead. In [14], researchers suggested an improvement that minimizes the number of persist operations. The idea is to have a *dirty* bit inside the value, which indicates whether the value was explicitly persisted or not. When modifying the value, the *dirty* flag is set to *true.* All readers, first check this flag and proceed as follows:

- If a flag is set: a reader uses compare-and-swap to clear the bit and then persists the value.
- If a flag is not set: the value is for sure persistent, and the reader can proceed.

One other problem which arises in concurrent applications is keeping the global state consistent. For example, to guarantee that calculating the number of elements in a data structure has O(1) time complexity, a *size* variable is usually used. In a concurrent case, a global variable cannot be simply modified in a transaction without taking a global lock. One possible solution is to make the global *size* volatile and introduce persistent, per-thread *size*. The global variable can be modified using atomic instructions while providing the number of elements in O(1) time. Persistent *sizes* can be summed up on recovery to restore the total amount of entries. Chapter 5 presents an implementation of this approach using the *enumerable_thread_specific* data structure from the libpmemobj-cpp library.

5. RELATED WORK

This chapter presents an overview of several research papers focused on designing data structures for Persistent Memory. While it's possible to take existing data structures developed for traditional storage and use them in Persistent Memory, this will not yield the best performance. The reason is that those data structures were designed for high latency block devices, and are not capable of taking full advantage of Persistent Memory byte addressability and low latency. On the other hand, data structures optimized for in-memory usage can achieve very low latency but were designed with the assumption that the memory is volatile. Those structures can be modified to guarantee consistency (for example, by introducing a transaction when a modification is needed). However, those modifications are usually very challenging and incur high performance or memory cost. One example of such an adaptation is [8]. This paper presented an approach for making a Red-Black Tree consistent even in case of system failures.

As described in the first chapter, Persistent Memory can differ from DRAM also in its read and write bandwidth. For example, if write bandwidth is lower than read bandwidth, reducing the number of writes (even at the price of more reads) can be beneficial for performance. Data structures designed for volatile memory usually assume the write and read bandwidth to be the same.

5.1. Overview of design primitives

In [11], the authors provided a comprehensive comparison between different design primitives for Persistent Memory data structures. The microbenchmarks which they designed show what kind of trade-offs each design incurs. Most of the implementation is generic and can be used to implement primitives for different data structures. Design primitives are mainly a method of storing key and value pairs. They consider sorted or unsorted approaches. In an unsorted case, there can also be some indirection mechanism. It can be implemented by a bitmap and a slot array - bitmap stores information whether an entry is valid, and slots store the sorted order of the keys. This approach was used in the design of B+-Tree by Shimin Chen et al. [12]. Another possibility is to use a hash map, which maps keys to their actual position in the array, as done in HiKV [10]. Another design choice that is investigated by Philipp Götze et al. is using only Persistent Memory vs. using a hybrid solution. In the hybrid solution, inner nodes are placed in DRAM and leaves in Persistent Memory.

To test the performance of different design primitives, the authors defined a set of micro-operations. Some of them are:

- Node search (finding a key-value pair within a node). The natural candidate - a sorted variant with a binary search does not perform very well on Persistent Memory. It has similar performance to linear search for small nodes. This is because of the necessity to jump between different cache lines. The best choice for performance is using a hashing approach – it provides the lowest latency.

- Tree traversal. As reported, adding one additional level to the tree increases the traversal latency by about 50-100ns for DRAM and about 400-500ns for Persistent Memory. It means that it is very beneficial to store the inner node in volatile memory and only leaf nodes in Persistent Memory. This hybrid approach also has one other advantage – splitting or merging nodes in DRAM do not have to be atomic. The program rebuilds the volatile state on each restart.
- Node insert. The unsorted variant provides the best performance as it consists only of writing new data at the end of the array and incrementing a counter. The sorted version is several times slower than any other solution for this operation because of the necessity to keep the ordering (a lot of writes and a consistency cost). All indirection approaches have similar performance, slightly worse than the unsorted one.
- Element erase. Unsorted and indirect methods show the best performance. In the case of the bitmap, the erase operation has to clear only a single bit. For the unsorted approach, it's necessary to replace the deleted element with the last entry and decrement the counter. The sorted approach, similarly as for insert, is slower by a large margin.

Authors concluded the paper with the following statements:

- A hybrid approach is highly recommended if recovery time is not a problem.
- Reading non-sequential data (in blocks less than 256B) is slow.
- Persistent Memory allocations are expensive – using a slab allocator might be a good idea.
- Node size should not be smaller than 256B

5.2 FPTree

Ismail Oukid et al. [20] implemented a concurrent, hybrid B-tree named FPTree. Lucas Lersch et al. [13] showed that this data structure achieves the best performance in most operations (lookup, insert, update, delete and scan were tested) compared to NVTree [21], wBtree [12], and BzTree [16]. FPTree achieves excellent performance by:

- Fingerprinting. FPTree stores keys in an unsorted array. Fingerprints are one-byte hashes of keys and are placed in the first cache-line of the leaf. When looking for a key, fingerprints are scanned first. This limits the number of probed keys to one on average.
- Selective persistence. Internal nodes of the B-tree are stored in DRAM and are rebuilt on recovery. Only leaves reside on Persistent Memory.
- Selective concurrency. Volatile and persistent parts of the data structure use different concurrency mechanisms. Inner nodes use Hardware Transactional Memory (HTM), and leaves use fine-grained locks. The reason for having two different methods is that HTM is incompatible with Persistent Memory. The CPU could detect a cache line flush as a conflict – this would abort the HTM transaction and result in fallback to locking in software.

- Amortized Persistent Memory allocations. FPTree allocates leaves in blocks. Freeing a leaf means putting into an internal pool for later reuse.

Results presented in [13] are another confirmation of conclusions made in 4.1. The selective persistence and minimizing the cost of keeping data consistent (for example, storing data unsorted and using techniques like fingerprinting) are beneficial for performance.

5.3 BzTree

Joy Arulraj et al. [16] presented a BzTree – a lock-free, main-memory B+-Tree that supports insert, read, update, delete, and range scan operations. To achieve lock freedom, it uses persistent multi-word compare-and-swap (PMwCAS) [14]. PMwCAS is a mechanism that allows changing multiple 8-byte words with persistence guarantees atomically. It is based on volatile multi-word compare-and-swap [15]. PMwCAS solves the visibility problem using a flush-on-read principle, as described in 2.4. PMwCAS is entirely lock-free and allows threads to cooperate. If there is a conflict (two threads try to modify the same value), one of them will help the other to complete. If setting some values using PMwCAS fails, then it is guaranteed that no other thread has seen those changes (even in case of power failure).

BzTree uses a technique known as epoch-based reclamation for reclaiming memory. When a thread wants to perform some operation, it first "joins" a current epoch that protects memory it accessed from being freed. After the procedure is over, it exits the epoch. When all active threads have joined, completed its operations, and exited the epoch, memory from that epoch can be reclaimed (no thread, after exiting can reference that memory). A separate background thread can perform the reclamation periodically, acting as a garbage collector.

The BzTree can work in volatile or persistent mode. In the former case, all nodes reside in Persistent Memory. It guarantees that all updates are persistent, and the recovery process is fast.

5.4. Write Optimized Radix Tree [9]

Write Optimized Radix Tree (WORT), Write Optimized Adaptive Radix Tree (WOART), and ART + CoW are three variants of Persistent Memory-optimized radix trees presented in [9]. Radix tree can be a good fit for Persistent Memory as it does not require any rebalancing operations. Benchmarks done by the authors of WORT, WOART, and ART + CoW show that those radix tree implantations perform better than some existing B+-Trees like FPTree [20] or NVTree [21] (which are also designed for Persistent Memory). However, the benchmarks were not performed on real Persistent Memory hardware (it was not available at that time) but instead using latency emulation. Moreover, the radix tree's implementation mentioned above [52] uses a regular volatile allocator, which can be orders of magnitude faster than a persistent one [13]. Given all that, performance results on PMEM might be significantly different from those reported by the authors.

5.5. Pronto [18]

Pronto is a library that allows modifying volatile data structures for pmem easily. The main idea of pronto is to have three entities – volatile image of the data structure, Asynchronous Semantic Logs (ASLs), and a persistent snapshot of the data. When a method to update a data structure is called Pronto does two things:

1. Performs the required operations on the volatile image
2. Logs method call to the ASL

Note that ASL does not contain information about specific bytes of the data structure (like in redo or undo logs) but only the type of the operation (insert, remove, etc.) and its arguments. Replaying stored operations after the crash is enough to perform a recovery. Updating volatile image and logging to ASL is done concurrently (a background thread is responsible for writing to ASL), which hides latency. Of course, an operation is considered completed only after both threads complete. Pronto also performs periodic snapshots of the volatile images to speed up the recovery and decrease the space needed for ASLs. The volatile image uses native, volatile pointers, so it must always reside at the same virtual address (which requires a specialized allocator).

Pronto is a promising approach – it can be easily used for concurrent data structures (if they are linearizable) and shows better latency than other existing solutions (e.g., pmemkv[19]). The problem, however, is that it introduces significant memory overhead. Each data structure has to have a full volatile copy.

5.6. Summary

Presented papers give several guidelines for programmers who wish to implement a Persistent Memory data structure. It's crucial to minimize write-intensive operations like tree rebalancing or inserting an element to a sorted array. Not only are those operations slower than on DRAM due to lower write bandwidth, but they also require using transactions or other techniques for keeping data consistent. Another critical aspect is minimizing the number of data structure's levels on Persistent Memory. Because of PMEM latency, each pointer jump introduces significant performance overhead. One way of resolving this is by storing some of the nodes in DRAM and restoring them on recovery. Researchers also suggest using at least 256B nodes in random workloads to achieve near-sequential performance.

6. PERSISTENT RADIX TREE

This chapter is dedicated to the analysis of the persistent radix tree that provides failure atomicity. It is one of the engines in the pmemkv database [19]. Its implementation is based on a volatile "critnib" tree from vmemcache [25]. As I have shown in the previous chapter, data structures on pmem should minimize the number of writes to pmem. A radix tree fits this requirement as there is no need for any rebalancing operations. Also, because of path compression, the number of levels in the radix tree can be kept low. The next section (6.1.) will describe the basic concepts behind the radix tree. After that, I will present the design of a persistent version of this tree. First, I will describe the layout and details of insert, search, and remove operations. After that, I will discuss the radix tree performance with respect to other pmemkv engines.

6.1. Radix tree overview

A radix tree (also radix trie or compact prefix tree) is a data structure that supports insertion, deletion, search operation, range query, and finding all elements with a common prefix. Insert, deletion, and search operations have *O(k)* time complexity, where k is the maximum length of all keys in the set [39]. The length is measured as *n/r*, where n is the number of key's bits, and r is the radix of the radix trie (fanout). It is a space-optimized version of trie (prefix tree) [39].

The radix tree has two types of nodes – internal and leaves. In most implementations, internal nodes contain only pointers to children (up to r children), and leaves contain values. Each edge between nodes has a label consisting of one or more bits. This label represents part of the key. Specifically, a label from root to node A is a common prefix of all keys in node N's subtree.

Figure 8 presents an example of a radix tree.

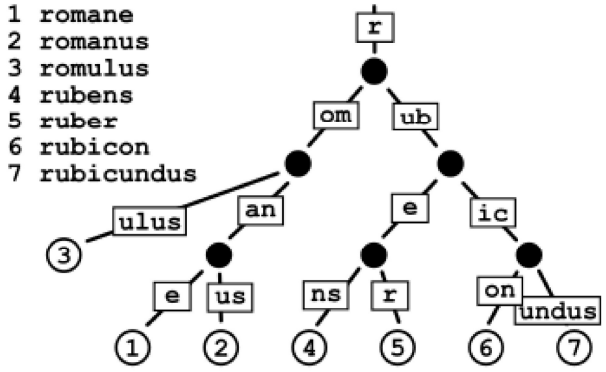

Figure 8 - An example of a radix tree. Source: https://en.wikipedia.org/wiki/Radix_tree

When looking for a specific key, a chunk of key's bits is compared with the labels from a node so that appropriate child can be selected. This is done on each level until a leaf is found, or it's not possible to continue (no child with a label matching the key). Depending on a variant, keys can

be either stored implicitly in the tree (as labels between nodes) or explicitly in leaves. In the former case, internal nodes can contain pointers to the appropriate part of the key.

In the radix tree, each internal node has at least two children. If removing a node would result in an internal node having only one child, this internal node has to be replaced by its only remaining child. This optimization (also called path compression) is what differentiates a radix tree from prefix trees.

Unlike traditional trees, the radix tree does not use comparison to locate elements. This means that the radix tree can only be used with keys which are strings or can be easily mapped to strings. For example, the *uint32_t* type from C/C++ might be used as a key type. The integer value must be treated as a sequence of 4 bytes (string). To keep the ordering between integer values, a radix tree must interpret the bytes in the correct order (an implementation must differentiate between little-endian and big-endian).

6.2. pmem::obj::radix_tree

The radix tree's implementation from libpmemobj-cpp is a variation of the radix tree, similar to the PATRICIA tree [40]. Internal nodes do not store any part of keys as labels. Instead, they contain information about a chunk (a group of bits) number that differentiates its subtrees. Moreover, in the libpmemobj-cpp implementation, internal nodes can be associated with values. It is necessary in order to allow storing keys, which are prefixes of each other. If data would only be stored in leaves, having both "a" and "aa" keys would not be possible. Additionally, both key and value can be stored within the same allocation as the leaf nodes. This minimizes the number of levels in the tree and hence improves performance

The API of *pmem::obj::radix_tree* is based on *std::map* API from C++ standard library. The full API description can be found here: https://pmem.io/libpmemobj-cpp/master/doxygen

6.2.1. Leaf node layout

The leaf node contains a pointer to the parent, the key, and the value. Value can be any C++ type that is suitable to be stored on PMEM. The key type has to be convertible to a sequence of bytes. Listing 11 presents the layout of leaf nodes on *pmem::obj::radix_tree*.

Listing 11 – Layout of leaf nodes on pmem::obj::radix_tree. Source: [24]

```
tagged_node_ptr parent;

Key key;

Value value;
```

6.2.2 Internal node layout

Listing 12 presents layout of internal nodes in *pmem::obj::radix_tree*.

Listing 12 – Layout of internal nodes on pmem::obj::radix_tree. Source: [24]

```
tagged_node_ptr parent;
tagged_node_ptr embedded_entry;
tagged_node_ptr child[SLNODES];

byten_t byte;
bitn_t bit;
```

The internal node contains:

- A pointer to the parent node (used for iteration)
- A pointer to the embedded entry. It is used to allow storing keys, which are prefixes of each other. If present, the embedded entry's key corresponds to a path from the root to the given internal node.
- An array of pointers to children. Each child points to a subtree that can be either a single leaf node or another internal node. The size of this array is a compile-time constant and depends on a radix of the radix tree.
- Byte and bit variables. Together are used to calculate the chunk (also called *NIB*), which differentiates the subtrees. This chunk is used to index the child array. The size of the chunk is equal to $\log_2(SLNODES)$ bits. To illustrate how the chunk is calculated, let's say we have a key = 0xABCD. For *byte* = 0, *bit* = 4 we have chunk = 0xA. For *byte* = 0, *bit* = 0 we have chunk = 0xB. The values of *n->byte* are increasing along with the depth of the tree.

Every pointer in the radix tree is tagged. The least significant bit specifies whether the pointer points to a leaf or to an internal node. All allocations from libpmemobj-cpp must be properly aligned, so it's safe to use that bit.

Figure 9 presents an example radix tree which holds 4 keys: "a", "ac", "acxxxy", "acxxxz". Internal nodes are colored blue, and the leaves are colored green.

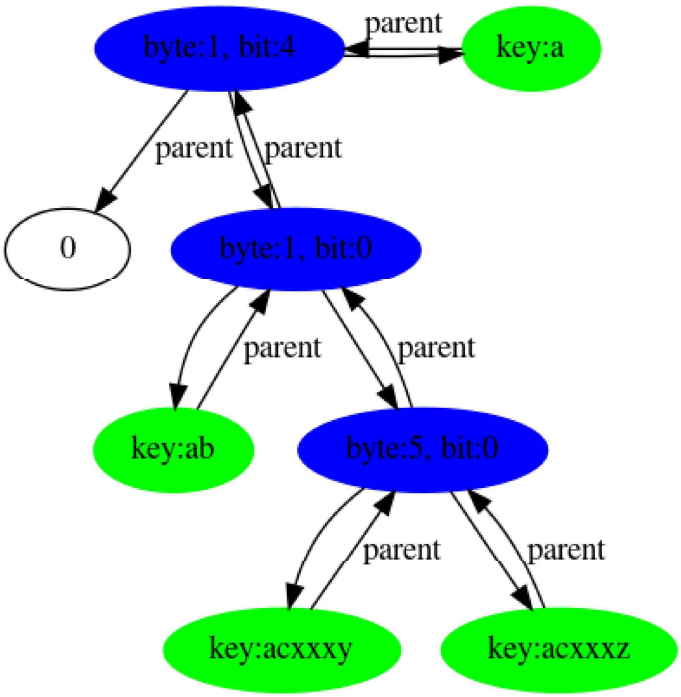

Figure 9 - Radix tree representation. Source: Own work.

As shown in figure 9, *pmem::obj::radix_tree* uses path compression. If several keys have some common parts ("xxx" in this example), they are omitted in the internal nodes. For example, the second (from top) internal node differentiates its subtrees by looking at the second byte (*byte == 1*). For the left subtree, the byte is equal to "b," and for the right subtree, it's equal to "c". However, instead of considering the third byte, the third internal leaf skips bytes from 1 to 4 and looks at the one that is different between leaves (*byte == 5*). Of course, it's not possible to tell what is the value of omitted bytes without descending to a leaf.

6.2.3. Failure atomicity

pmem::obj::radix_tree provides a failure atomicity guarantee by using libpmemobj transactions (described in previous chapters). If a crash occurs during the modifier method execution, all changes are discarded, and the radix tree is rolled back to the last consistent state. It's also possible to group several operations in a single transaction. This is shown in listing 13.

Listing 13 – Grouping radix tree modifiers in a single transaction. Source: own work.

```
pmem::obj::transaction::run(pop, [&]{
    radix->insert("a", "a");
    radix->insert("b", "b");

    radix->erase("c");
});
```

All node allocations and deallocations inside methods like insert or erase are performed by libpmemobj allocator. Every pointer modification is preceded by snapshotting the old value. This is done inside the *tagged_node_ptr* assignment operator. A possible implementation of such an operator is shown in listing 14.

Listing 14 – Powefail-safe assignment. Source: own work.

```
tagged_node_ptr& operator=(const tagged_node_ptr& rhs) {
    pmem::obj::transaction::snapshot(&this->offset);
    this->offset = rhs.offset;
}
```

Such an approach makes *pmem::obj::radix_tree* code similar to a volatile data structure implementation. All transactional-related functions are hidden in the type implementation. libpmemobj-cpp native pointer type (*persistent_ptr*) also snapshots the data automatically.

6.2.4. Search operation

Searching for an element is very straightforward. Listing 15 presents an implementation of a find() method, which returns iterator to the element (or past-the-end iterator if the element is not found [53]).

Listing 15 – Implementation of find method. Source: [24]

```
auto n = root;
auto child_slot = &root;
while (n && !n.is_leaf()) {
    if (path_length_equal(key.size(), n))
        child_slot = &n->embedded_entry;
    else if (n->byte > key.size())
        return end();
    else
        child_slot =
            &n->child[slice_index(key[n->byte], n->bit)];

    n = *child_slot;
}

if (!n)
    return end();

if (!keys_equal(key, bytes_view(n.get_leaf()->key())))
    return end();

return const_iterator(child_slot, &root);
```

The *find* method consists of a loop that runs until a leaf node or null pointer is found. It starts at the root. At each level key's length is compared with the *n->byte.* If the key size is larger than the *n->byte* (a third condition in the loop), a new child from *n->child* array is picked based on a value

of *key[n->byte]* and *n->bit*. *n->byte* and *n->bit* determine the chunk of bits representing a "label" from *n* to its *child*.

If a key length is equal to the path from the root to *n* (first conditional), *n->embedded_entry* is potentially the looked-for element (length of the path from the root to *n* is equal to key size). Otherwise, If the second condition is true, then the algorithm has descended too far. *n* represents a subtree of elements whose keys have length at least *n->byte*. If the looked-for item has not been found yet, there is no such item in the tree.

The last step in the algorithm is to compare the found element with the key. Since *pmem::obj::radix_tree* uses path compression, the candidate node might not be the looked-for element (the parts of the key which were compressed might differ).

6.2.5. Insert method

If the tree is empty, it's enough to allocate a leaf and set a root pointer to point to that leaf. Otherwise, it is necessary to descend to the bottom of the tree. This is done by an algorithm similar to the one used in find() operations. The difference is that the algorithm does not stop if it encounters a null pointer or a node with too high *byte* value. Instead, it continues to any leaf in the subtree. This is done to find out the actual edge labels (some of them are stored implicitly due to a path compression). Figure 10 presents which parts of a label are known without knowing the key (it is the same tree as in figure 9).

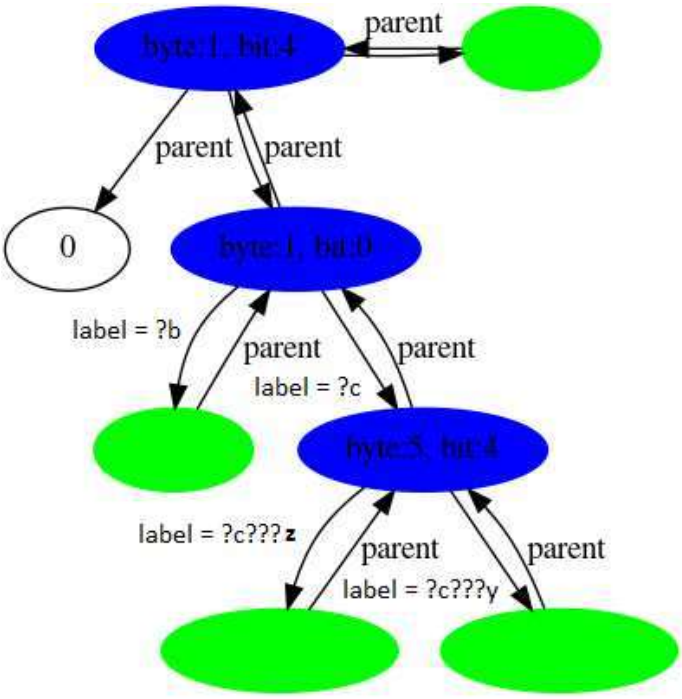

Figure 10 - Radix tree representation. Source: own work

The characters represented by question marks can only be known after descending to a leaf. If the two bottom leaves are "acxxxy" and "acxxxz" then it's obvious that the label "?b" is actually "ab" and "?c" is "ac".

After finding out the labels, the algorithm can proceed with finding a proper place for the new element. There are several cases which should be analyzed separately.

1. Inserting an element, which is a prefix of an already existing element (let's call it L). In this case, the algorithm follows a path to the element L but stops when it encounters a node with a *byte* value larger or equal to the new element's key size (path from the root to this node has a length equal or larger than the key size). If the *n->byte* (where is the found node) is equal to the key size, then a new leaf is created and pinned to *n->embedded_entry*. If *n->byte* is larger, then it's necessary to allocate a new parent node. The latter case is shown in figure 11.

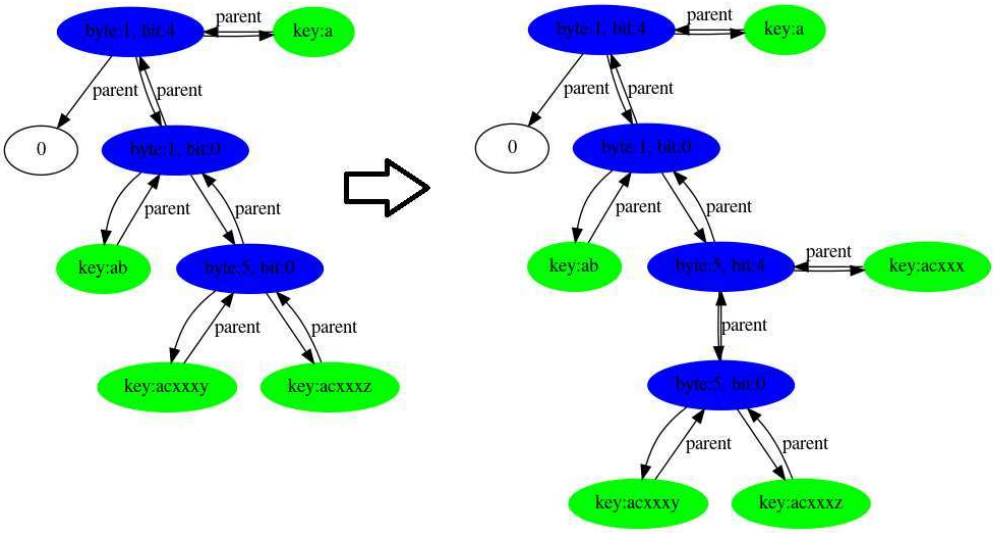

Figure 11- Radix tree representation. Source: own work

2. Inserting an element whose prefix already exists in the tree. Similarly to the case 1, the algorithm follows the path to a leaf (in this case, the leaf contains a key which is a prefix of the new element). After that, it converts the leaf into a node (allocates a new internal node, and sets *embedded_entry* pointer the old leaf). The new element is added as a

child leaf to that new internal node. This is presented in figure 12.

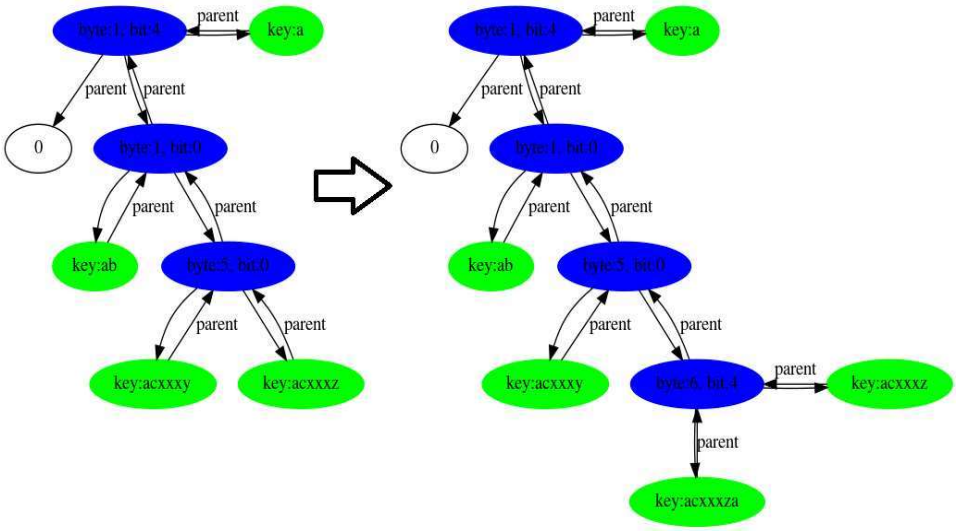

Figure 12 - Radix tree representation. Source: own work

3. Inserting an element that differs from all elements existing in the tree at some position.
    a. There is a free slot in the child array which matches the element exactly. In that case, the algorithm just finds the slot and puts the new leaf. Note that the new leaf can have an arbitrarily long postfix. The tree in figure 13 would look the same for "acxxxw" as for "acxxxw_some_postfix". The only thing that matters is the chunk of bits at the position specified by *n->byte* and *n->bit*.

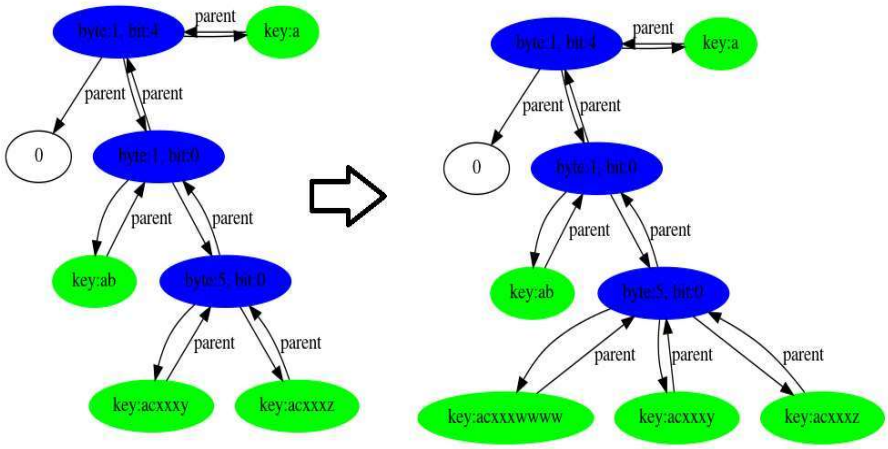

Figure 13 - Radix tree representation. Source: own work

    b. There is no free slot in the child array (path is compressed, and the element differs from the labels). This case is shown in figure 14. Due to a path compression, there is no internal node that could become a parent of the new

"acxyxw" element. This means the algorithm has to break the compression and allocate one more internal node. The compressed path consisting of one label "cxxx" is split into two labels (two levels) "cx" and "yxw"/"xx".

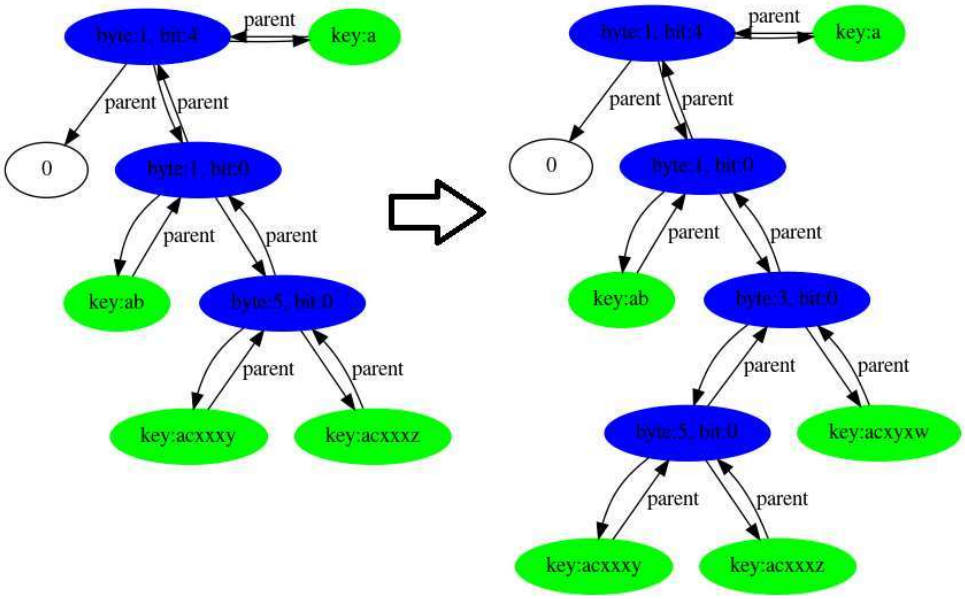

Figure 14 - Radix tree representation. Source: own work

6.2.6. Remove operation

Remove operation undoes the changes made by an insert operation. For example, consider an example in figure 15, where the element with key "acxyxw" is removed. First, an algorithm finds the leaf which contains the wanted element. Then, it frees the leaf and clears a pointer to that leaf in a parent's child array. If, at this point, the parent contains only one child, it can be removed. This situation is presented in figure 15. After deleting the leaf, a node with *byte*=3 would have only one child. Because of that, it is also removed and replaced by its only child.

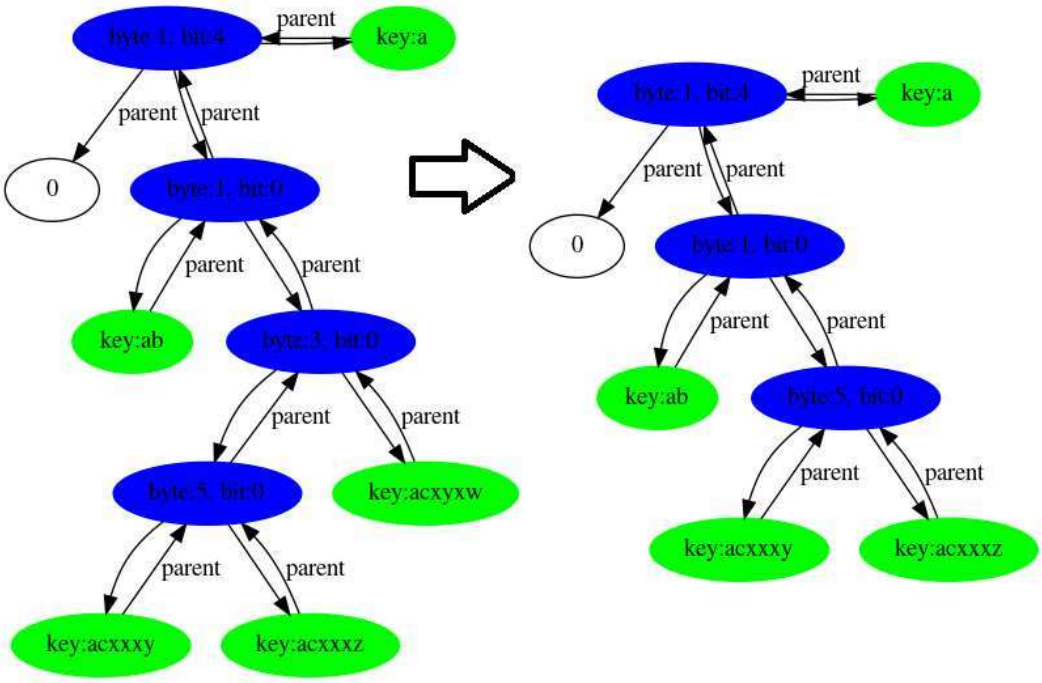

Figure 15 - Radix tree representation. Source: own work

6.3. Performance evaluation

This section provides an assessment of a persistent radix tree performance when compared to other pmemkv's [19] persistent engines. The performance measurements were performed using pmemkv-tools [41] utility, which contains a modified version of the level-db benchmark [42]. The numbers presented here were obtained from running the benchmarks on a regular DRAM (not on PMEM). The absolute performance measured on Persistent Memory will be different. However, because of using the same allocator and the same methods for ensuring data consistency, relative performance should be very similar. Moreover, all persistent pmemkv engines (including radix tree) use 256B aligned nodes whenever possible. Depth of the tested data structures is also similar, which reduces the impact of latency difference between PMEM and DRAM.

6.4.1. Pmemkv persistent engines

The radix tree was compared with stree and csmap engines.

Stree – a persistent, single-threaded, and sorted engine backed by a B+ tree. The version used in experiments had a degree equal to 32 (each node can have up to 32 children).

Csmap – a persistent, concurrent, and sorted engine backed by a skip list. The underlying skip list is implemented in the libpmemobj-cpp repository [43]. It is based on a volatile implementation

proposed by M. Herlihy et al. [44]. This engine's performance is expected to be lower than stree because of the use of locks (during inserts) and atomic read operations (during reads).

Both engines use *pmem::obj::string* as key and value type. *pmem::obj::string* implements small string optimization for values of size less or equal to 23 bytes. Whenever a key or value is shorter than 23 bytes, it is stored within the same allocation as the node (instead of keeping a pointer to a string allocated on persistent heap).

6.4.2. Setup

The benchmarks were performed on a machine equipped with Intel® Core™ i7-8700K at 3.7 GHz and 16GB RAM on Ubuntu 19.10 (kernel 5.3.0-64-generic). The experiments were run using a command presented in listing 16.

Listing 16 – The command used to run pmemkv benchmarks—source: Own work.

```
PMEM_IS_PMEM_FORCE=1 ./pmemkv_bench --engine=engine_name --
db_size_in_gb=6 --benchmarks=fillrandom,readrandom,fillseq,readseq --
histogram=1 --num=5000000
```

PMEM_IS_PMEM_FORCE environment variable ensures that libpmemobj does not use msync calls (which are typically required to ensure consistency on non-dax file systems).

6.4.3. Results

Figures 16, 17, 18, 19, and 20 show the performance of radix, stree, and csmap engines in single-threaded fillrandom, readrandom, and readseq workloads. Key and value sizes are equal to 16 bytes and 100 bytes, respectively. The number of elements inserted or queried was equal to 5000000. Fillrandom benchmark inserts a specified number of randomly generated keys to the empty database. Readrandom was executed just after fillrandom and searches for randomly generated keys. Readseq benchmark was run after filling the database in sequential order (keys from 0 to 5000000). For figures 16 and 17, higher values are better. For figures 18, 19 and 20, lower is better.

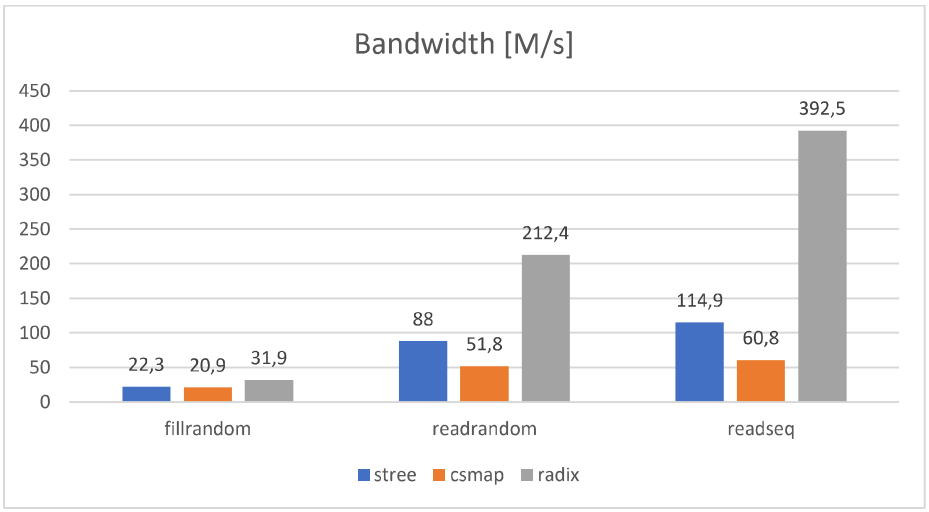

Figure 16 - Bandwidth comparison (key size = 16, value size = 100). Source: own work.

As shown in figure 16, the bandwidth of the radix tree in the fillrandom benchmark is about 1.5 times better than stree or csmap. In readrandom and readseq, the difference is even more significant – more than two times for readrandom and ~3.4 times for readseq. Different methods of finding the key can explain such a big difference. Both stree and csmap use comparisons to find elements (there are $\log_2(n)$ key comparisons on average). In contrast, the radix tree uses key bytes directly to find an element's position in the tree (there is only one key comparison). It's also essential to notice that stree and csmap store keys and values inside pmem::obj::string. Figure 17 presents bandwidth for fillrandom, readrandom, and readseq when key and value sizes are both equal to 16 bytes (both key and value can take advantage of small string optimization).

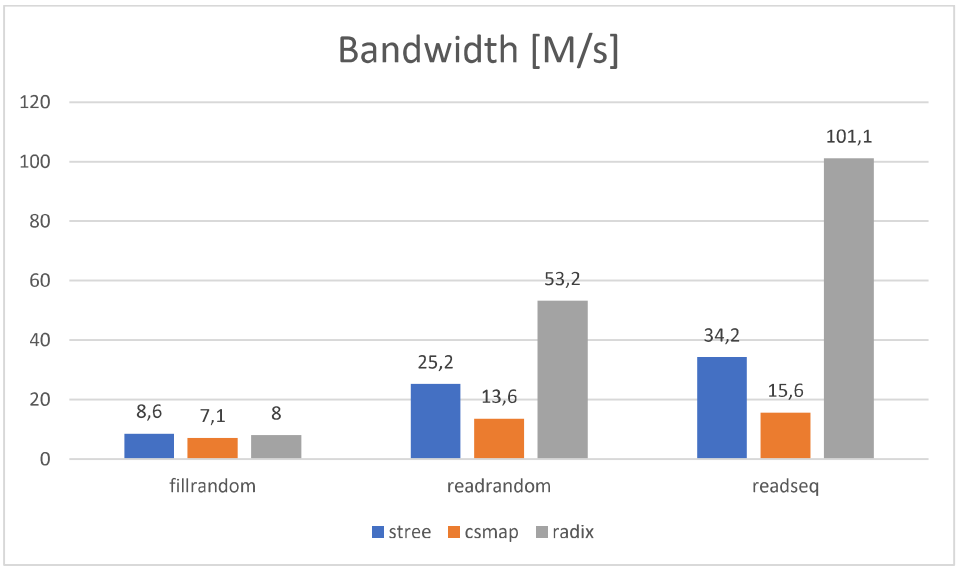

Figure 17 - Bandwidth comparison (key size = 16, value size = 16). Source: own work.

As seen in figure 17, the fillrandom benchmarks show the slight advantage of a stree engine. However, in the case of readrandom and readseq radix engine still prevails.

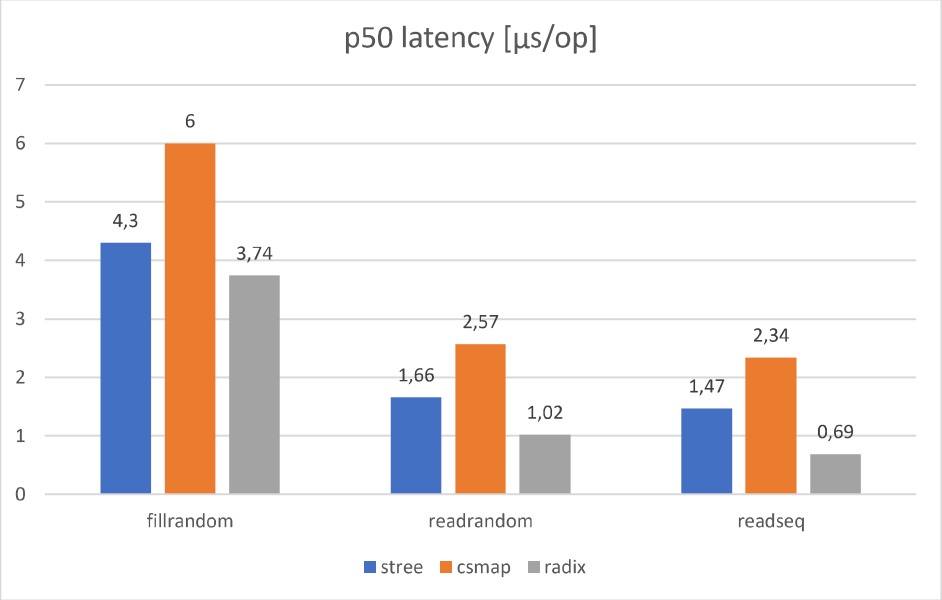

Figure 18 – P50 latency comparison. Source: Own work.

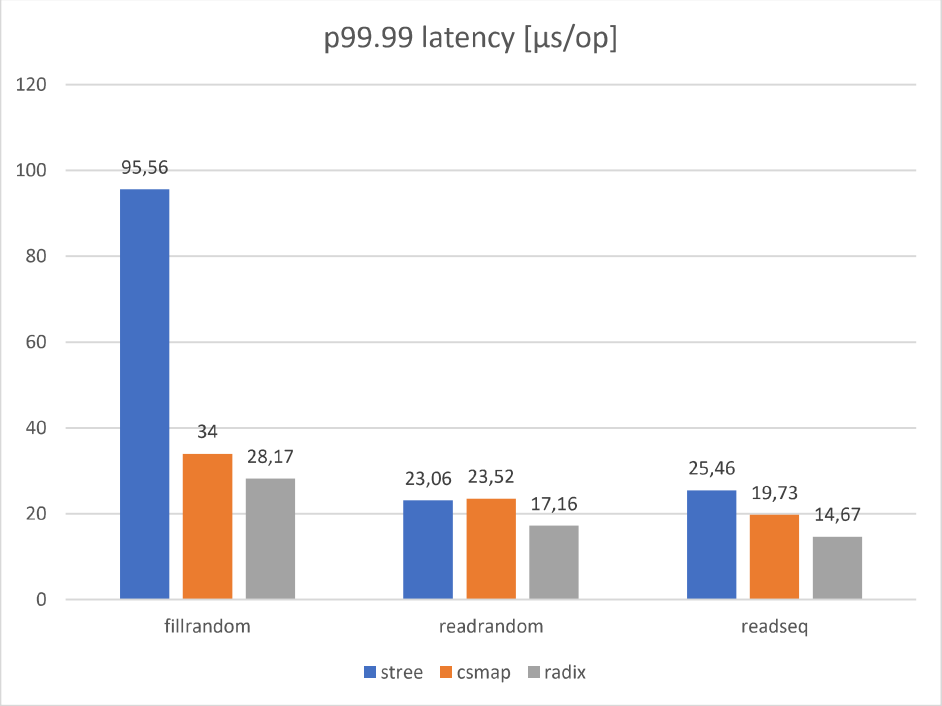

Figure 19 - P99.99 latency comparison. Source: Own work.

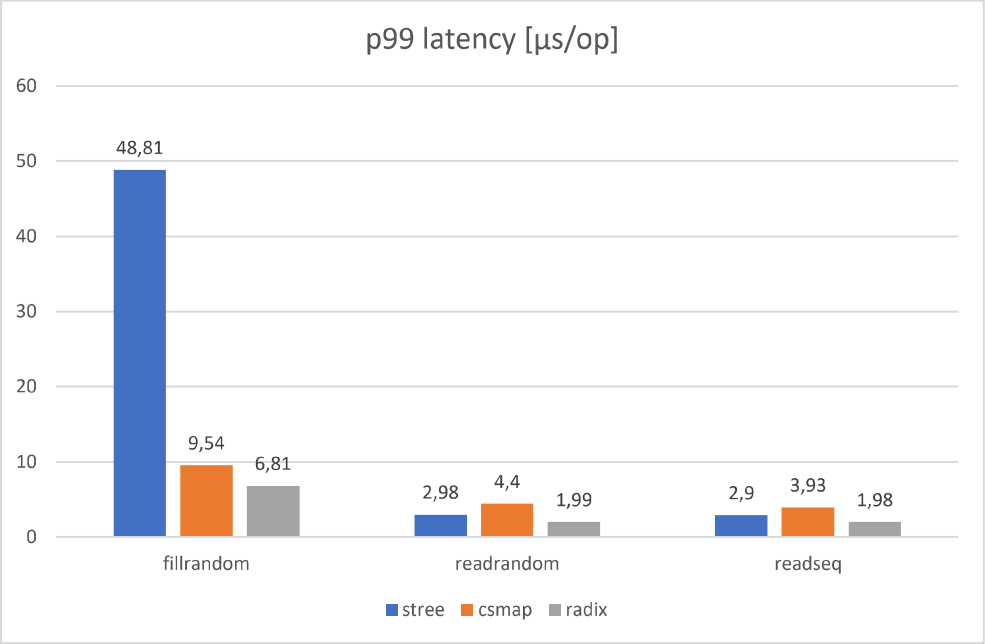

Figure 20 - P99 latency comparison. Source: Own work.

Figures 18, 19, and 20 present latency results. The radix tree shows the best latency for all benchmarks. The most significant difference can be seen for p99. The latency of the radix tree for fillrandom is more than seven times lower than for stree and 1.4 times better than for csmap. The high latency values for stree are most probably caused by a node split and merge operations. It can be controlled by lowering stree degree value but with a negative impact on bandwidth.

6.5. Summary

The radix tree is a data structure that fits very well with Persistent Memory. Lack of costly rebalancing operations results in a very low p99.99 and p99 latency when compared to a B+ tree. Also, limiting the number of full key comparisons results in much higher read bandwidth than comparator-based data structures like B+ tree or a skip list. The radix tree performs better than other pmemkv sorted engines in both write and read benchmarks.

7. TESTING PERSISTENT MEMORY DATA STRUCTURES

This chapter describes the methods of testing data structures and algorithms developed for Persistent Memory. It focuses on verifying data consistency. Several tools make this job easier. In this chapter, I will briefly introduce two of them – pmemcheck and pmreorder. Both of them were used to verify the radix tree implementation's correctness. Appropriate tests can be found in libpmemobj-cpp and pmemkv repositories.

7.1. Pmemcheck

Pmemcheck is a tool available in a modified version of Valgrind [45]. It is similar to memcheck (a tool for discovering memory leaks and other memory-related bugs). Its main objective is to validate the correctness of stores made to Persistent Memory. All PMDK libraries are already instrumented with pmemcheck, so any code using PMDK can be run under this tool without any modifications. To execute a program under pmemcheck, the following command can be used (presented in listing 17).

Listing 17 – The command to run any application under valgrind. Source: https://pmem.io/2015/07/17/pmemcheck-basic.html

```
$ valgrind --tool=pmemcheck [valgrind options] <your_app> [your_app
options]
```

Pmemcheck can detect various persistent programming bugs. Some of them will be described in the following sections.

7.1.1. Non-persistent stores

Let's consider a simple C++ program that writes to bytes to Persistent Memory. The code is shown in listing 18.

Listing 18 – Writing to pmem without persist—source: Own work.

```
18  struct root {
19      char data[1024];
20  };
21
22  int
23  main(int argc, char *argv[])
24  {
25      const char *path = argv[1];
26
27      pmem::obj::pool<root> pop;
28      pop = pmem::obj::pool<root>::open(path, "example");
29
30      pop.root()->data[0] = 0;
31      pop.root()->data[1] = 1;
32
33      pop.close();
34  }
```

Executing the code from listing 18 will result in the pmemcheck error presented in listing 19.

Listing 19. Pmemcheck output. Source: Valgrind pmemcheck output.

```
==662== pmemcheck-1.0, a simple persistent store checker
==662== Copyright (c) 2014-2020, Intel Corporation
==662== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright
info
==662== Command: example test
==662==
==662==
==662== Number of stores not made persistent: 2
==662== Stores not made persistent properly:
==662== [0]    at 0x40350F: main (example.cpp:30)
==662==     Address: 0x57b5450 size: 1     state: DIRTY
==662== [1]    at 0x403533: main (example.cpp:31)
==662==     Address: 0x57b5451 size: 1     state: DIRTY
==662== Total memory not made persistent: 2
==662== ERROR SUMMARY: 2 errors
```

The output from the pmemcheck shows that the data was not persisted correctly. The log contains addresses and sizes of dirty stores, as well as a full stacktrace. A fixed program is shown in listing 20 (it has an additional pmemobj_persist call).

Listing 20. Fixed code. Source: Own work.

```
18  struct root {
19      char data[1024];
20  };
21
22  int
23  main(int argc, char *argv[])
24  {
25      const char *path = argv[1];
26
27      pmem::obj::pool<root> pop;
28      pop = pmem::obj::pool<root>::open(path, "example");
29
30      pop.root()->data[0] = 0;
31      pop.root()->data[1] = 1;
32      pmemobj_persist(pop.handle(), pop.root()->data, 2);
33
34      pop.close();
35  }
```

Pmemcheck also works well with pmemobj transactions. Consider the code in Listing 21, which modifies a variable inside of a transaction.

Listing 21. Not adding data to a transaction. Source: own work.

```
18  struct root {
19      int var;
20  };
21
22  int
23  main(int argc, char *argv[])
24  {
25      const char *path = argv[1];
26
27      pmem::obj::pool<root> pop;
28      pop = pmem::obj::pool<root>::open(path, "example");
29
30      pmem::obj::transaction::run(pop, [&] {
31          pop.root()->var = 10;
32      });
33
34      pop.close();
35  }
```

This code, similarly as one from listing 18, does not persist the data correctly. In this case, the pmemcheck will also report that the store is not a part of a transaction. The pmemcheck output is presented in listing 22.

Listing 22. Pmemcheck output. Source: Valgrind pmemcheck output.

```
==712== Number of stores not made persistent: 1
==712== Stores not made persistent properly:
==712== [0]    at 0x4048F5: main::$_0::operator()() const
(example.cpp:31)
==712==    by 0x4047AC: std::_Function_handler<void (),
main::$_0>::_M_invoke(std::_Any_data const&) (std_function.h:300)
==712==    by 0x40532D: std::function<void ()>::operator()() const
(std_function.h:690)
==712==    by 0x404DE4: void
pmem::obj::transaction::run<>(pmem::obj::pool_base&, std::function<void
()>) (include/libpmemobj++/transaction.hpp:423)
==712==    by 0x4045B3: main (example.cpp:30)
==712==     Address: 0x57b5450 size: 4     state: DIRTY
==712== Total memory not made persistent: 4
==712==
==712== Number of stores made without adding to transaction: 1
==712== Stores made without adding to transactions:
==712== [0]    at 0x4048F5: main::$_0::operator()() const
(example.cpp:31)
==712==    by 0x4047AC: std::_Function_handler<void (),
main::$_0>::_M_invoke(std::_Any_data const&) (std_function.h:300)
==712==    by 0x40532D: std::function<void ()>::operator()() const
(std_function.h:690)
==712==    by 0x404DE4: void
pmem::obj::transaction::run<>(pmem::obj::pool_base&, std::function<void
()>) (include/libpmemobj++/transaction.hpp:423)
==712==    by 0x4045B3: main (example.cpp:30)
==712==     Address: 0x57b5450 size: 4
```

```
==712== ERROR SUMMARY: 2 errors
```

To fix the problem, one can use a persistent property class (pmem::obj::p), which will take care of snapshotting the data when modified inside a transaction. The fixed program is presented in listing 23.

Listing 23 – Fixed program. Source: Own work.

```
18  struct root {
19      pmem::obj::p<int> var;
20  };
21
22  int
23  main(int argc, char *argv[])
24  {
25      const char *path = argv[1];
26
27      pmem::obj::pool<root> pop;
28      pop = pmem::obj::pool<root>::open(path, "example");
29
30      pmem::obj::transaction::run(pop, [&] {
31          pop.root()->var = 10;
32      });
33
34      pop.close();
35  }
```

7.1.2. Overwrites

Pmemcheck can report overwriting the data before it is persisted. To enable this option, Valgrind must be run with `--mult-stores=yes`. Listing 24 shows an example program that writes to variable data[0] two times before calling pmemobj_persist.

Listing 24. Overwriting data. Source: Own work.

```
18  struct root {
19      char data[1024];
20  };
21
22  int
23  main(int argc, char *argv[])
24  {
25      const char *path = argv[1];
26
27      pmem::obj::pool<root> pop;
28      pop = pmem::obj::pool<root>::open(path, "example");
29
30      pop.root()->data[0] = 1;
31      pop.root()->data[0] = 2;
32      pmemobj_persist(pop.handle(), pop.root()->data, 2);
33
34      pop.close();
35  }
```

In this program, if a crash happens before calling pmemobj_persist value of data[0] is unspecified. It can be 0 (initial value), 1, or 2. Overwriting a persistent variable like this is often a bug. Pmemcheck will report an error, as shown in listing 25.

Listing 25. Pmemcheck output. Source: Valgrind pmemcheck output.

```
==791== Number of stores not made persistent: 0
==791==
==791== Number of overwritten stores: 1
==791== Overwritten stores before they were made persistent:
==791== [0]    at 0x40351F: main (example.cpp:30)
==791==     Address: 0x57b5450 size: 1     state: DIRTY
==791== ERROR SUMMARY: 1 errors
```

7.1.4. Memory added to different transactions

Pmemcheck can also help to catch errors related to concurrent programming. If a program adds some memory regions to two (or more) transactions running in different threads simultaneously, data corruption might occur. Pmemcheck will detect this problem. To see why adding memory to multiple transactions is problematic, let's look at figure 21.

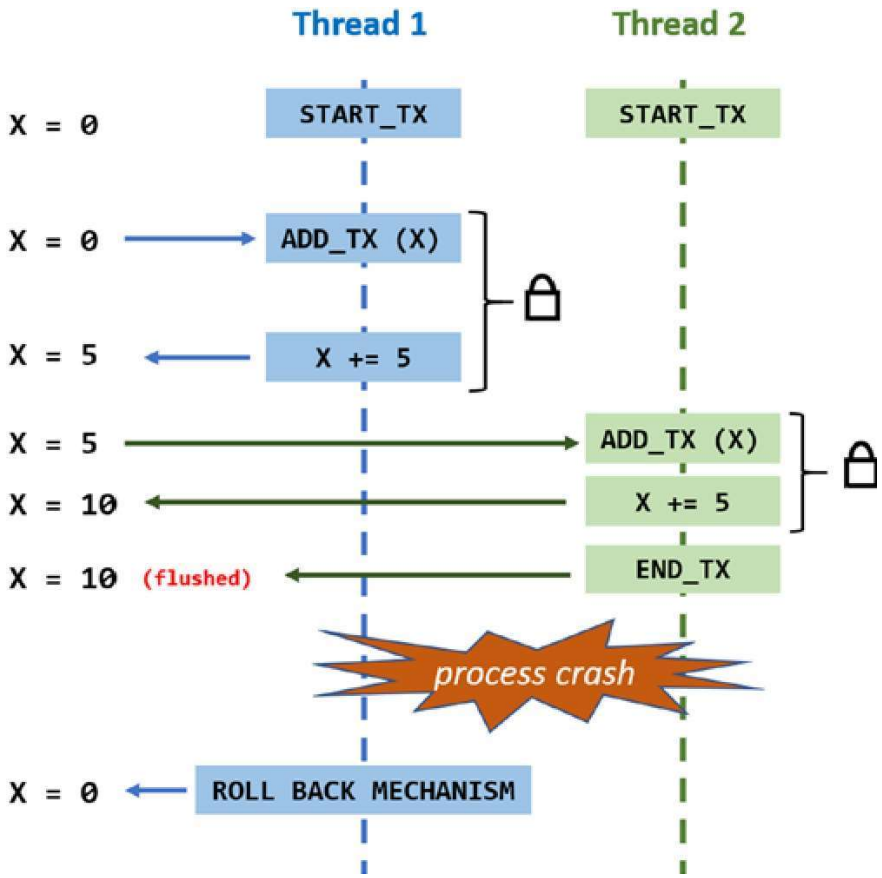

Figure 21 - Adding memory to two transactions running simultaneously. Source: Scargall, Steve: Programming Persistent Memory A Comprehensive Guide for Developers [1]

In figure 21, both threads snapshot variable X to a transaction and modify it. The second thread succeeds and commits the transaction (X has now value 10). After that, because of a process crash, thread number 1 is interrupted. On recovery, the rollback mechanism will restore the initial value of (as it was at the time of snapshotting). This means that the update of an already committed transaction from the second thread will be lost. A program that contains such a bug is presented in listing 26.

Listing 26. Adding data to multiple transactions. Source: own work.

```
19  struct root {
20      pmem::obj::p<int> var;
21  };
22
23  int
24  main(int argc, char *argv[])
25  {
26      const char *path = argv[1];
27
28      pmem::obj::pool<root> pop;
29      pop = pmem::obj::pool<root>::open(path, "example");
```

```
    30
    31      std::thread t1([&]{
    32          pmem::obj::transaction::run(pop, [&]{
    33              pop.root()->var = 5;
    34          });
    35      });
    36
    37      std::thread t2([&]{
    38          pmem::obj::transaction::run(pop, [&]{
    39              pop.root()->var = 10;
    40          });
    41      });
    42
    43      t1.join();
    44      t2.join();
    45
    46      pop.close();
    47  }
```

When running this code under pmemcheck, one will get an error, such as in listing 27.

Listing 27 – Pmemcheck output. Source: Valgrind pmemcheck output (lines trimmed for readability).

```
==940== Number of stores not made persistent: 0
==940==
==940== Number of overlapping regions registered in different transaction
s: 1
==940== Overlapping regions:
==940== [0]    at 0x4866ABC: pmemobj_tx_add_snapshot.constprop.0 ...
==940==    by 0x4866F47: pmemobj_tx_add_common.constprop.0 ...
==940==    by 0x4868233: pmemobj_tx_xadd_range_direct ...
==940==    by 0x408A3F: void pmem::detail::conditional_add_to_tx<pmem::ob
j:...
==940==    by 0x4089BA: pmem::obj::p<int>::swap(pmem::obj::p<int>&)...
==940==    by 0x40887C: pmem::obj::p<int>::operator=(pmem::obj::p<int> co
ns...
==940==    by 0x40692C: main::$_1::operator()() const::{lambda()#1}::oper
at...
==940==    by 0x4067BC: std::_Function_handler<void (), main::$_1::operat
or...
==940==    by 0x406EDD: std::function<void ()>::operator()()...
==940==    by 0x407C44: void pmem::obj::transaction::run<>...
==940==    by 0x40664B: main::$_1::operator()() const ...
==940==    by 0x4065FC: void std::__invoke_impl<void, main::...
==940==     Address: 0x57b5450  size: 4 tx_id: 3
==940==    First registered here:
==940== [0]'   at 0x4866ABC: pmemobj_tx_add_snapshot.constprop.0 ...
==940==    by 0x4866F47: pmemobj_tx_add_common.constprop.0 ...
==940==    by 0x4868233: pmemobj_tx_xadd_range_direct ...
==940==    by 0x408A3F: void pmem::detail::conditional_add_to_tx<pmem::ob
j:...
==940==    by 0x4089BA: pmem::obj::p<int>::swap(pmem::obj::p<int>&) ...
==940==    by 0x40887C: pmem::obj::p<int>::operator=(pmem::obj::p<int> co
ns...
```

```
==940==    by 0x405F8C: main::$_0::operator()() const::{lambda()#1}::oper
a...
==940==    by 0x405E1C: std::_Function_handler<void (), main::$_0::operat
or...
==940==    by 0x406EDD: std::function<void ()>::operator()() const (std_f
un...
==940==    by 0x407C44: void pmem::obj::transaction::run<>(pmem::obj::poo
l...
==940==    by 0x405CAB: main::$_0::operator()() const ...
==940==    by 0x405C5C: void std::__invoke_impl<void, main::$_0>(std::__i
n...
==940==     Address: 0x57b5450  size: 4 tx_id: 2
==940== ERROR SUMMARY: 1 errors
```

7.2. Pmreroder

Pmreorder is a tool that can help find bugs related to stores ordering. As described in chapter 2, a program must issue a flush (CLFLUSHOPT, CLFLUSH, or CLWB) instruction to persist some data. Since CLFLUSHOPT and CLWB are asynchronous, they should be followed by an SFENCE instruction (which acts as a synchronization point). Let's look at figure 22.

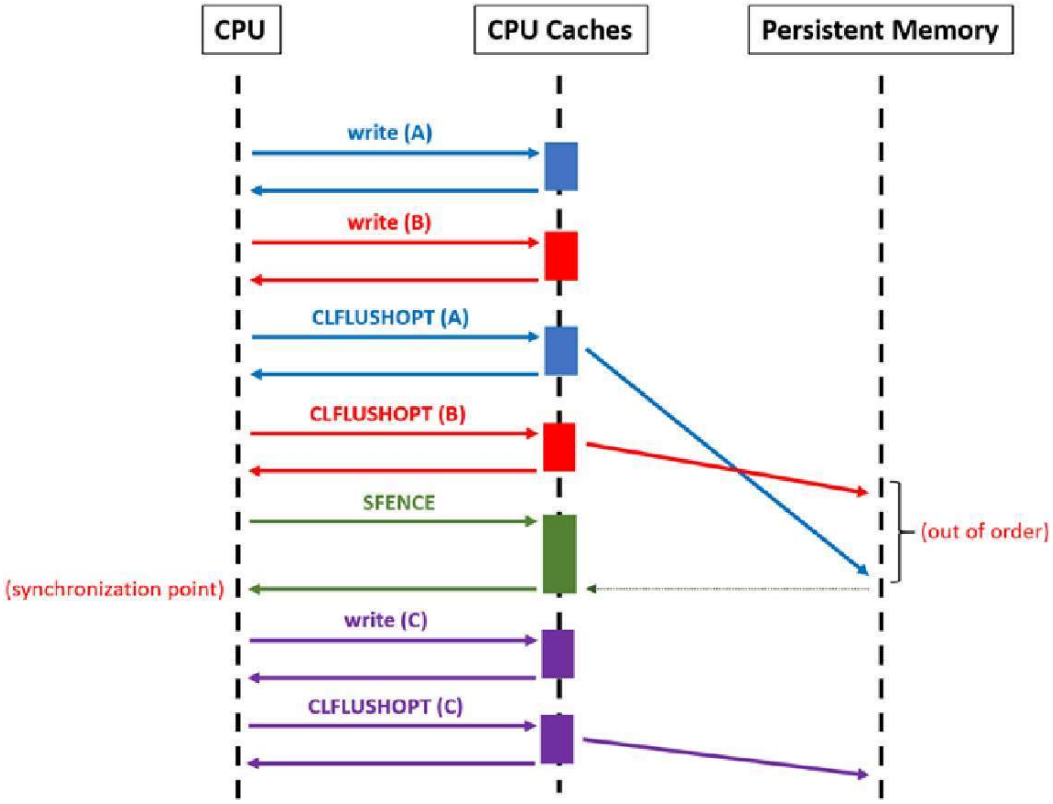

Figure 22 - SFENCE usage example. Source: Scargall, Steve: Programming Persistent Memory A Comprehensive Guide for Developers 2020 [1]

In figure 22, a program executes two writes, followed by two CLFLUSHOPT instructions and an SFENCE instruction. After the fence, the program issues additional write and CLFLUSHOPT. In this example, SFENCE guarantees that all writes made before the fence are made persistent before write(C) – CPU is not allowed to reorder write(A) or write(B) with write(C). However, since there is no SFENCE between write(A) and write(B), the CPU can reorder A and B. Not enforcing a proper ordering of stores (by using SFENCE) can result in data corruption. To find such problems, it is necessary to record stores made by an application (by using pmemcheck as described in [46]). Then, the list of stores is passed to pmreorder along with a checker program. Pmreorder reorders stores between fences (it simulates CPU reordering) and issues them. After that, a checker program is executed to verify data consistency. This procedure can be repeated multiple times for different reorderings.

Listing 28 shows a program contains a bug related to stores reordering.

Listing 28 – Missing persist. Source: Own work.

```
22  #define LAYOUT "pmreorder"
23
24  namespace nvobj = pmem::obj;
25
26  struct Data {
27      char str[128];
28      bool valid;
29  };
30
31  struct root {
32      nvobj::persistent_ptr<Data> ptr;
33  };
34
35  void
36  run_inconsistent(nvobj::pool<root> &pop)
37  {
38      auto r = pop.root();
39
40      memset(r->ptr->str, 1, 128);
41      r->ptr->valid = true;
42
43      pmemobj_persist(pop.handle(), r->ptr->str, 128);
44      pmemobj_persist(pop.handle(), &r->ptr->valid, sizeof(bool));
45  }
46
47  void
48  check_consistency(nvobj::pool<root> &pop)
49  {
50      auto r = pop.root();
51
52      if (r->ptr->valid) {
53          for (int i = 0; i < 128; i++)
54              assert(r->ptr->str[i] == 1);
55      }
56  }
57
58  int
```

```
59  main(int argc, char *argv[])
60  {
61      const char *path = argv[2];
62
63      nvobj::pool<root> pop;
64
65          if (argv[1][0] == 'o') {
66              pop = nvobj::pool<root>::open(path, LAYOUT);
67
68              check_consistency(pop);
69          } else if (argv[1][0] == 'c') {
70              pop = nvobj::pool<root>::create(path, LAYOUT,
71                              PMEMOBJ_MIN_POOL * 20,
72                              S_IWUSR | S_IRUSR);
73
74              nvobj::transaction::run(pop, [&] {
75                  pop.root()->ptr = nvobj::make_persistent<Data>();
76                  });
77          } else if (argv[1][0] == 'i') {
78              pop = nvobj::pool<root>::open(path, LAYOUT);
79
80              run_inconsistent(pop);
81          }
82
83      pop.close();
84  }
```

In this example, run_inconsistent() function calls persist after modifying both str and valid field. However, the check_consistency method assumes that if a valid field is set, memory under str will be filled with ones. If such a program were run under pmreorder (exact instructions can be found on pmem.io [46]), it would produce a log, as shown in listing 29.

Listing 29. Pmemcheck output. Source: pmreorder output (trimmed down to few lines).

```
example: /opt/workspace/tests/example/example.cpp:54: void
check_consistency(nvobj::pool<root> &): Assertion `r->ptr->str[i] == 1'
failed.
Aborted (core dumped)

-- Pmreorder:
WARNING:pmreorder:File /tmp/example/testfile inconsistent
WARNING:pmreorder:Call trace:
Store [0]:
    by0x4057ED: run_inconsistent(pmem::obj::pool<root>&)
(tests/example/example.cpp:41)
    by0x405DB1: main (tests/example/example.cpp:80)

WARNING:pmreorder:File /tmp/example/testfile inconsistent
WARNING:pmreorder:Call trace:
Store [0]:
    by0x4057ED: run_inconsistent(pmem::obj::pool<root>&)
(tests/example/example.cpp:41)
    by0x405DB1: main (tests/example/example.cpp:80)
Store [1]:
```

```
    by 0x4DB9F8F: __memset_avx2_unaligned_erms (memset-vec-unaligned-
erms.S:193)
    by 0x4057DB: run_inconsistent(pmem::obj::pool<root>&)
(tests/example/example.cpp:40)
    by 0x405DB1: main (tests/example/example.cpp:80)
```

The pmreorder output contains the program's stderr content (assertion failure information) and a stacktrace, specifying which stores were performed before calling the check_consistency method. To fix the program from listing 28, pmemobj_persist should be called just after modifying str variable via memset. The corrected version of the run_inconsistent function is shown in listing 30.

Listing 30. Fixed run_inconsistent function. Source: Own work.

```
35  void
36  run_inconsistent(nvobj::pool<root> &pop)
37  {
38      auto r = pop.root();
39
40      memset(r->ptr->str, 1, 128);
41
42      pmemobj_persist(pop.handle(), r->ptr->str, 128);
43      r->ptr->valid = true;
44
45      pmemobj_persist(pop.handle(), &r->ptr->valid, sizeof(bool));
46  }
```

7.3. Summary

Ensuring data consistency in a Persistent Memory aware application is not an easy task. However, some tools can help significantly in catching bugs, which could lead to data loss or corruption. This chapter presented two of them – pmemcheck and pmreorder. There are also other tools worth mentioning. The first one is Intel Inspector – Persistence Inspector [47], which has capabilities similar to pmemcheck and pmreorder. The second one is PMTest [48], a framework for crash-consistency testing.

8. SUMMARY

This thesis presented an overview of Persistent Memory technology. I showed that creating efficient Persistent Memory applications requires a new programming model. Such a model was introduced by SNIA. Along with it, new tools and libraries, such as PMDK, emerged. The new model and unique characteristics of Persistent Memory hardware present a lot of new challenges for programmers. In this thesis, I gave an overview of several methods for guaranteeing data consistency, which is one of the biggest concerns. Chapter six showed that it's possible to implement an efficient data structure based on the SNIA model and PMDK. I also presented how Persistent Memory applications can be tested.

I think that Persistent Memory will see broad adoption in the industry. Although the new programming model is challenging, Persistent Memory can be used to build very high-performance systems. A recent example of that is a solution based on Intel DAOS (Distributed Asynchronous Object Store) combined with Intel® Optane™ Persistent Memory, which set a new world record and is now (as of August 11, 2020) ranked No.1 for file system performance worldwide [49].

LITERATURE

1. Scargall, Steve: Programming Persistent Memory A Comprehensive Guide for Developers 2020
2. https://pmem.io/ [access: 05.04.2020]
3. Joseph Izraelevitz et al. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module 9 Aug 2019
4. Intel® 64 and IA-32 Architectures Software Developer's Manual, https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-1-manual.pdf [access 05.04.2020]
5. https://pmem.io/pmdk/manpages/linux/master/libpmem/libpmem.7.html [access: 05.04.2020]
6. Alexander van Renen et al. Persistent Memory I/O Primitives https://db.in.tum.de/~vanrenen/papers/nvm_stats.pdf [access: 05.04.2020]
7. Intel® 64 and IA-32 Architectures Optimization Reference Manual https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf [access: 05.04.2020]
8. Chundong Wang et al. Persisting RB-Tree into NVM in a Consistency Perspective https://www.researchgate.net/publication/323407018_Persisting_RB-Tree_into_NVM_in_a_Consistency_Perspective [access: 05.04.2020]
9. Se Kwon Lee et al. https://www.usenix.org/system/files/conference/fast17/fast17-lee.pdf [access: 05.04.2020]
10. Fei Xia HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems https://www.usenix.org/system/files/conference/atc17/atc17-xia.pdf [access: 05.04.2020]
11. Philipp Götze et al. Data Structure Primitives on Persistent Memory: An Evaluation https://arxiv.org/pdf/2001.02172.pdf [access: 05.04.2020]
12. Shimin Chen et al. Persistent B+-Trees in Non-Volatile Main Memory http://www.vldb.org/pvldb/vol8/p786-chen.pdf [access: 05.04.2020]
13. Lucas Lersch et al. Evaluating Persistent Memory Range Indexes http://www.vldb.org/pvldb/vol13/p574-lersch.pdf [access: 05.04.2020]
14. Tianzheng Wang et al. Easy Lock-Free Indexing in Non-Volatile Memory http://justinlevandoski.org/papers/mwcas.pdf [access: 05.04.2020]
15. Timothy L. Harris et al. A Practical Multi-Word Compare-and-Swap Operation https://www.cl.cam.ac.uk/research/srg/netos/papers/2002-casn.pdf
16. Joy Arulraj BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory http://www.vldb.org/pvldb/vol11/p553-arulraj.pdf [access: 05.04.2020]
17. Justin J. Levandoski et al. The Bw-Tree: A B-tree for New Hardware Platforms https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/bw-tree-icde2013-final.pdf [access: 05.04.2020]

18. Amirsaman Memaripour et al. Pronto: Easy and Fast Persistence for Volatile Data Structures https://dl.acm.org/doi/pdf/10.1145/3373376.3378456 [access: 05.04.2020]
19. https://github.com/pmem/pmemkv [access: 05.04.2020]
20. Ismail Oukid et al. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory https://wwwdb.inf.tu-dresden.de/misc/papers/2016/Oukid_FPTree.pdf
21. J. Yang et al. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems https://www.usenix.org/system/files/conference/fast15/fast15-paper-yang.pdf [access: 05.04.2020]
22. https://en.wikipedia.org/wiki/Data-oriented_design [access: 05.04.2020]
23. https://pmem.io/pmdk/manpages/linux/master/libpmemobj/libpmemobj.7.html [access: 05.04.2020]
24. https://pmem.io/libpmemobj-cpp/ [access: 05.04.2020]
25. https://github.com/pmem/vmemcache/blob/master/src/critnib.c [access: 05.04.2020]
26. Accelerate Your Apache Spark with Intel Optane DC Persistent Memory, https://databricks.com/session/accelerate-your-apache-spark-with-intel-optane-dc-persistent-memory [access: 05.04.2020]
27. Uwe Heinz, SAP HANA and Persistent Memory https://blogs.sap.com/2020/01/30/sap-hana-and-persistent-memory/ [access: 05.04.2020]
28. Aleksandar Dragojevic et al. No compromises: distributed transactions with consistency, availability, and performance https://pdos.csail.mit.edu/6.824/papers/farm-2015.pdf [access: 05.04.2020]
29. Jim Handy, An NVDIMM Primer (Part 1 of 2) https://thessdguy.com/an-nvdimm-primer-part-1-of-2/ [access: 05.04.2020]
30. https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm [access: 05.04.2020]
31. Brian Beeler, Intel Optane DC Persistent Memory Module (PMM) https://www.snia.org/education/what-is-persistent-memory [access: 05.04.2020]
32. Chunyang Hui, Enabling Persistent Memory in the Storage Performance Development Kit (SPDK) https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html [access: 05.04.2020]
33. https://pl.wikipedia.org/wiki/Hibernate [access: 05.04.2020]
34. Michael Stonebraker, Joseph M. Hellerstein What Goes Around Comes Around https://people.cs.umass.edu/~yanlei/courses/CS691LL-f06/papers/SH05.pdf [access: 05.04.2020]
35. Charles Lamb et.al, The Objectstore database system https://web.eecs.umich.edu/~jag/eecs584/papers/objectstore.pdf [access: 05.04.2020]

36. Micheal J. Carey David J. DeWitt, A data model, and query language for EXODUS https://www.researchgate.net/publication/234813686_A_data_model_and_query_language_for_EXODUS [access: 05.04.2020]
37. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.45.2209&rep=rep1&type=pdf [access: 05.04.2020]
38. https://en.wikipedia.org/wiki/Schema_evolution [access: 05.04.2020]
39. https://en.wikipedia.org/wiki/Radix_tree [access: 05.04.2020]
40. Donald R. Morrison, PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric https://dl.acm.org/doi/10.1145/321479.321481 [access: 05.04.2020]
41. https://github.com/pmem/pmemkv-tools [access: 05.04.2020]
42. https://github.com/google/leveldb/blob/master/benchmarks/db_bench.cc [access: 05.04.2020]
43. https://github.com/pmem/libpmemobj-cpp/blob/master/include/libpmemobj%2B%2B/container/detail/concurrent_skip_list_impl.hpp [access: 05.04.2020]
44. Maurice Herlihy et al. A Provably Correct Scalable Concurrent Skip List https://www.cs.tau.ac.il/~shanir/nir-pubs-web/Papers/OPODIS2006-BA.pdf [access: 05.04.2020]
45. https://github.com/pmem/valgrind [access: 05.04.2020]
46. https://pmem.io/2019/02/04/pmreorder-basics.html [access: 05.04.2020]
47. https://techdecoded.intel.io/developer-tools/intel-inspector-with-persistence-memory-debugger/ [access: 05.04.2020]
48. Sihang Liu et al. PMTest: A Fast and Flexible Testing Framework for
49. Persistent Memory Programs https://www.sihangliu.com/docs/liu_asplos19.pdf [access: 05.04.2020]
50. https://newsroom.intel.com/articles/intel-optane-persistent-memory-daos-solution-sets-world-record/ [access: 05.04.2020]
51. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html [access: 05.04.2020]
52. NVM Programming Model (NPM) https://www.snia.org/sites/default/files/NVMProgrammingModel_v1.pdf [access: 05.04.2020]
53. https://github.com/SeKwonLee/WORT[access: 05.04.2020]
54. https://en.cppreference.com/w/cpp/container/map/end [access: 05.04.2020]

LIST OF FIGURES

LIST OF FIGURES
Figure 1 - Storage hierarchy. Source: https://www.snia.org/cation/what-is-persistent-memory 6
Figure 2 - SNIA Programming Model. Source: https://www.nextplatform.com/2018/02/07/momentum-gathers-persistent-memory-preppers/ … 9
Figure 3 - CPU Cache hierarchy. Source: https://pmem.io/2019/12/19/performance.html 10
Figure 4 - Undo log. Source: https://pmem.io/2020/03/26/performance-2.html 18
Figure 5 - Redo log. Source: https://pmem.io/2020/03/26/performance-2.html 19
Figure 6 - Versioning. Source: Steve Scargall, A Comprehensive Guide for Developers, Chapter 11 [1] ……. 21
Figure 7 - Insert operation. Source: Steve Scargall, A Comprehensive Guide for Developers, Chapter 11 [1] ……..…….. 22
Figure 8 - An example of a radix tree. Source: https://en.wikipedia.org/wiki/Radix_tree. ……29
Figure 9 - Radix tree representation. Source: Own work. …… 32
Figure 10 - Radix tree representation. Source: Own work. ……..…….. 34
Figure 11- Radix tree representation. Source: Own work. ……35
Figure 12 - Radix tree representation. Source: Own work. ……..…….. 36
Figure 13 - Radix tree representation. Source: Own work. ……..…… 36
Figure 14 - Radix tree representation. Source: Own work. ……..…….. 37
Figure 15 - Radix tree representation. Source: Own work. ……..…….. 38
Figure 16 - Bandwidth comparison (key size = 16, value size = 100). Source: own work. ……. 40
Figure 17 - Bandwidth comparison (key size = 16, value size = 16). Source: own work. ……... 40
Figure 18 – P50 latency comparison. Source: Own work. ……..…… 41
Figure 19 - P99.99 latency comparison. Source: Own work. …….. 41
Figure 20 - P99 latency comparison. Source: Own work. …… 32
Figure 21 - Adding memory to two transactions running simultaneously. Source: Scargall, Steve: Programming Persistent Memory A Comprehensive Guide for Developers [1] ……... 48
Figure 22 - SFENCE usage example. Source: Scargall, Steve: Programming Persistent Memory A Comprehensive Guide for Developers 2020 [1] ……..…… 50