

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
GRADUATION THESIS**

Ускорение записи на жёсткие диски с помощью NVRAM / Acceleration of writes to hard drives with NVRAM

Обучающийся / Student Довжик Лев Игоревич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования

Группа/Group M42381

Направление подготовки/ Subject area 01.04.02 Прикладная математика и информатика

Образовательная программа / Educational program Программирование и искусственный интеллект 2021

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Магистр

Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан	
Довжик Лев Игоревич	
14.05.2023	

(эл. подпись/ signature)

Довжик Лев
Игоревич

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
14.05.2023	

(эл. подпись/ signature)

Аксенов
Виталий
Евгеньевич

(Фамилия И.О./ name
and surname)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**ЗАДАНИЕ НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ /
OBJECTIVES FOR A GRADUATION THESIS**

Обучающийся / Student Довжик Лев Игоревич

Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования

Группа/Group M42381

Направление подготовки/ Subject area 01.04.02 Прикладная математика и информатика

Образовательная программа / Educational program Программирование и искусственный интеллект 2021

Язык реализации ОП / Language of the educational program Русский

Статус ОП / Status of educational program

Квалификация/ Degree level Магистр

Тема ВКР/ Thesis topic Ускорение записи на жёсткие диски с помощью NVRAM / Acceleration of writes to hard drives with NVRAM

Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

Основные вопросы, подлежащие разработке / Key issues to be analyzed

Research objective: Development of NVRAM cache for acceleration of synchronous writes to storage devices

Research tasks:

- a) Study of architectural features of NVRAM and basic principals of writing programs with it?
- b) Develop of NVRAM cache for acceleration synchronous writes to append-only files
- c) Check power failure tolerance and benchmark latency of developed cache

Форма представления материалов ВКР / Format(s) of thesis materials:

Source code, presentation, thesis paper

Дата выдачи задания / Assignment issued on: 10.02.2023

Срок представления готовой ВКР / Deadline for final edition of the thesis 24.05.2023

Характеристика темы ВКР / Description of thesis subject (topic)

Тема в области фундаментальных исследований / Subject of fundamental research: нет / not

Тема в области прикладных исследований / Subject of applied research: да / yes

СОГЛАСОВАНО / AGREED:

Руководитель ВКР/
Thesis supervisor

Документ подписан	
Аксенов Виталий Евгеньевич	
14.05.2023	

Аксенов
Виталий
Евгеньевич

(эл. подпись)

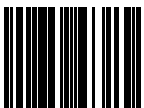
Задание принял к
исполнению/ Objectives
assumed BY

Документ подписан	
Довжик Лев Игоревич	
14.05.2023	

Довжик Лев
Игоревич

(эл. подпись)

Руководитель ОП/ Head
of educational program

Документ подписан	
Парфенов Владимир Глебович	
17.05.2023	

Парфенов
Владимир
Глебович

(эл. подпись)

**Министерство науки и высшего образования Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО
ITMO University**

**АННОТАЦИЯ
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
SUMMARY OF A GRADUATION THESIS**

Обучающийся / Student Довжик Лев Игоревич
Факультет/институт/кластер/ Faculty/Institute/Cluster факультет информационных технологий и программирования
Группа/Group M42381
Направление подготовки/ Subject area 01.04.02 Прикладная математика и информатика
Образовательная программа / Educational program Программирование и искусственный интеллект 2021
Язык реализации ОП / Language of the educational program Русский
Статус ОП / Status of educational program
Квалификация/ Degree level Магистр
Тема ВКР/ Thesis topic Ускорение записи на жёсткие диски с помощью NVRAM / Acceleration of writes to hard drives with NVRAM
Руководитель ВКР/ Thesis supervisor Аксенов Виталий Евгеньевич, PhD, науки, Университет ИТМО, институт прикладных компьютерных наук, доцент (квалификационная категория "ординарный доцент")

**ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ
DESCRIPTION OF THE GRADUATION THESIS**

Цель исследования / Research goal

Development of NVRAM cache for acceleration of synchronous writes to storage devices

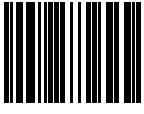
Задачи, решаемые в ВКР / Research tasks

1) Study of architectural features of NVRAM and basic principals of writing programs with it 2) Develop of NVRAM cache for acceleration synchronous writes to append-only files 3) Check power failure tolerance and benchmark latency of developed cache

Краткая характеристика полученных результатов / Short summary of results/findings

Successfully managed to develop cache for append-only files using NVRAM, that accelerates synchronous writes and recovers data after power failure

Обучающийся/Student

Документ подписан	
Довжик Лев Игоревич	
14.05.2023	

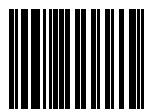
(эл. подпись/ signature)

Довжик Лев
Игоревич

(Фамилия И.О./ name
and surname)

Руководитель ВКР/
Thesis supervisor

Документ подписан
Аксенов Виталий Евгеньевич
14.05.2023



Аксенов
Виталий
Евгеньевич

(эл. подпись/ signature)

(Фамилия И.О./ name
and surname)

CONTENTS

INTRODUCTION	5
1. Review of a subject area	7
1.1. NVRAM	7
1.1.1. The architecture of NVRAM	8
1.1.2. NVRAM modes	10
1.2. Interaction with hardware	15
1.2.1. Device drivers	15
1.2.2. Operating system	15
1.2.3. System calls	18
1.2.4. File system	21
1.3. Databases	25
1.3.1. Transactions	28
Conclusions on Chapter 1	29
2. General description of the library	31
2.1. Features and limitations	31
2.2. Technologies used	32
2.2.1. libpmem	32
2.2.2. syscall_intercept	34
2.3. Library implementation	37
2.3.1. General design	37
2.3.2. Parameters passing	38
2.3.3. Intersection subtleties	39
2.3.4. Initial interception	41
2.3.5. Intercepted syscalls	41
2.3.6. Handler architecture	48
Conclusions on Chapter 2	54
3. Analysis of library	56
3.1. Performance	56
3.2. Correctness	64
Conclusions on Chapter 3	67
CONCLUSION	68
REFERENCES	69
APPENDIX A. Comparing libpmem to Linux mmap	72

INTRODUCTION

In recent years, the demand for high-performance computing systems has increased rapidly, with applications ranging from scientific simulations to data analytics. One critical aspect of such systems is the ability to efficiently handle large amounts of data, especially when it comes to a persistent storage. Synchronous writes to disk are an essential feature for many applications that require durability and consistency, but they can be a significant bottleneck in terms of performance.

Non-volatile random-access memory (NVRAM) is a promising technology that bridges the gap between volatile main memory and non-volatile storage. NVRAM provides the performance close to one of dynamic random access-memory (DRAM) used as main memory while also retaining data in case of a power failure or system crash, making it an ideal candidate for use in high-performance computing systems. By leveraging the capabilities of NVRAM, it may be possible to accelerate synchronous writes to disk, thereby improving the performance of applications that rely on them. This is especially relevant for append-only files, which are used as foundation of modern database management systems in form of database journal. This journal is responsible for durability of data stored in database and other its transactional properties. It also often limits latency of the write queries to the database.

However, NVRAM has two main disadvantages: firstly, its cost per GIB is very much close to the cost of ordinary volatile dynamic random-access memory, which makes it difficult to just replace standard storage devices with it, and, secondly, its capacity. Despite it being larger than the amount of dynamic random-access memory one could install in their platform, is still very limited compared to hard disk drives or even solid state drives.

Last aspect is the ability to easily integrate NVRAM-based solutions to your system. Despite the fact, that there are already solutions such as for example libpmemlog [15], which address issues mentioned in previous paragraph, they require significant changes in source code of programs that are run on systems with NVRAM. But it is often not easy to introduce such drastic changes in ones applications and redeploy them. It even gets worse when you don't have an access to source code of programs you use, because they are distributed as binary executables or libraries and maintainers are rarely eager to support unconventional hardware. However, for an end user it is reasonable to expect same level of

simplicity such as by just replacing all hard disk drives with solid state drives, otherwise, an adoption of this technology on mass scale becomes quite questionable.

Therefore, the following goals need to be achieved in order to complete the work:

- Get familiar with NVRAM and its architectural features.
- Get familiar with programming concepts used to develop application for NVRAM.
- Develop a library for caching synchronous writes to append-only with following properties:
 - No changes to end users' applications are needed to use this library.
 - Library is suitable to be used in multi-threaded environment.
 - No explicitly persisted data is lost in case of power failure.
- Test performance and safety guaranties of developed library

This thesis is structured as follows:

- The first chapter contains a review of the subject area. We describe an architecture of NVRAM and different modes it can operate in. We review how applications interact with hardware and which abstractions are used to do so in an application to storage devices. Lastly, we examine basic principals used in database management systems in order to archive consistency and durability to look for possible applications of this work
- The second chapter contains a detailed description of a developed library with explanation of its guaranties and limitations.
- The third chapter contains the details of how to test the library on real world application and its durability guaranties.

CHAPTER 1. REVIEW OF A SUBJECT AREA

1.1. NVRAM

Non-volatile random-access memory (NVRAM, also known as PMEM — persistent memory) is a type of memory technology that combines the speed of volatile random-access memory (RAM) with the persistence of non-volatile storage. This feature makes it a promising candidate for use in high-performance computing systems that require both high-speed data access and data durability. The specifics of NVRAM are the following:

- Unlike traditional volatile memory, which loses its contents when power is lost, NVRAM can retain data even when the power is turned off, which can be used to recover system after unexpected failure.
- NVRAM supports fast random-access to individual bytes of NVRAM similar to volatile random-access memory. This allows to develop more efficient algorithm than require persistence compared to approaches used with classical storage devices, because it is much more easier to place and keep consistent complicated data structures.
- Despite memory itself being non-volatile it is paired with volatile memory such as process registers or even its own caches that used in order to boost its performance, for example, for repeated reads of unchanged data. This means that the results of recent computations (which are often stored in process registers) or data which not fully flushed from cache will be lost. In order to prevent loss of cached data NVRAM allows to explicitly drain all caches to a non-volatile storage.
- NVRAM flushes its data in cache lines and guaranties that such writes will be atomic. It means that either the whole cache line is written to non-volatile or nothing is written. You cannot write half or quarter of a cache line. However, the writing a bigger amount of data is non-atomic and out of order, so if a power failure occurs before all data is written you can find any subset of a written data on NVRAM after the recovery.
- Current NVRAM implementations performance does not scale with its capacity the same way it happens with solid state drives, which means there is a very tangible limit of how much NVRAM could be put into system before it starts perform on par with classical storage devices.

Its advantages in mixed read/write random workloads, that are more typical for device with emphases on its random access, are shown in [23] and could be seen on figure 1:

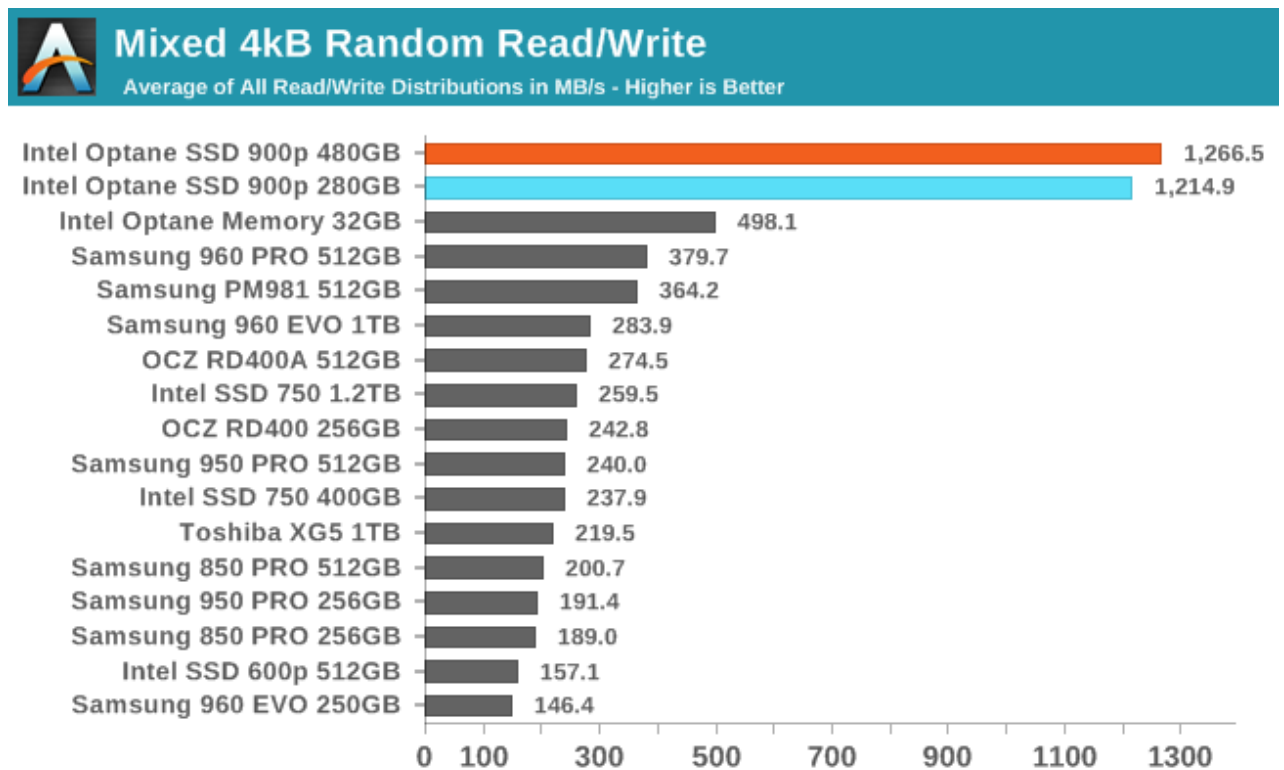


Figure 1 – Mixed random reads/writes performance benchmarks

1.1.1. The architecture of NVRAM

The architecture of NVRAM can vary depending on the specific technology used. One common type of NVRAM is based on a combination of DRAM and flash memory. In this architecture, the NVRAM module includes both DRAM chips and a flash memory chip. The DRAM chips are used as a cache for frequently accessed data, while the flash memory is used as a persistent store for less frequently accessed data. When data is written to the NVRAM module, it is first written to the DRAM cache, and then asynchronously written to the flash memory to ensure persistence. This type of NVRAM could be connected to computer via DDR bus or PCIe bus depending on particular implementation.

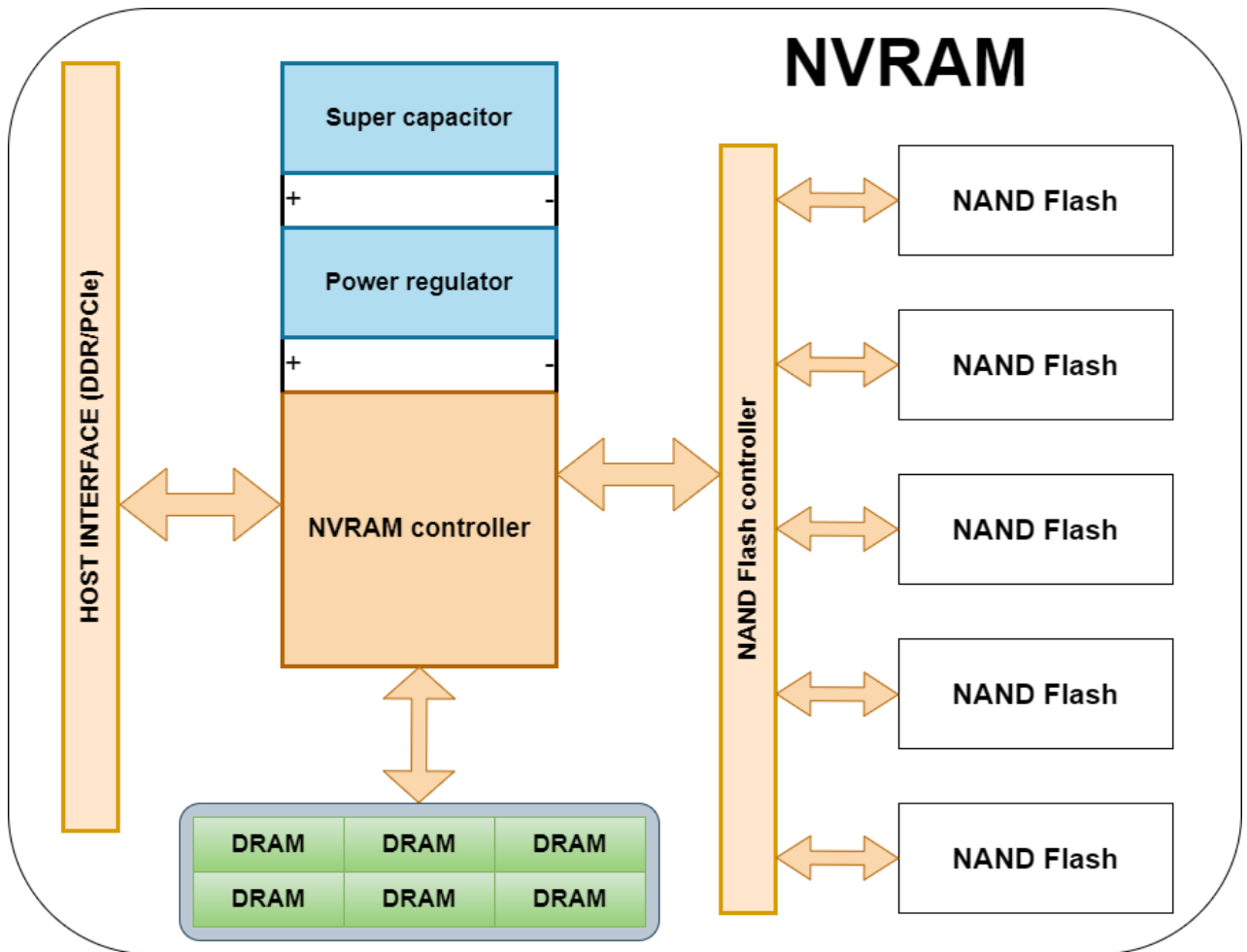


Figure 2 – General architecture of flash based NVRAM

Another type of NVRAM is based on a technology called Phase Change Memory (PCM). PCM is a type of non-volatile memory that uses the change in the state of a material between a crystalline and amorphous state to store data. PCM-based NVRAM modules work by storing data in a matrix of cells that can be switched between crystalline and amorphous states by applying electrical pulses. Like DRAM-based NVRAM, PCM-based NVRAM can offer high-speed access to data while also providing persistence. PCM-based devices support 100 million write cycles which leads to much slower degradation compared to flash-based devices where one sector can withstand about 5 thousand write and require sophisticated logic in controllers in order to perform wear leveling spread writes across many physical sectors [26]. Additionally, the resistivity of a memory element in PCM is more stable compared to flash memory, that “leaks” its charge over time, which leads to data corruption and loss, and is expected to hold data intact for 300 years at the normal working temperature, however, the main drawback is its sensitivity to temperature which af-

fects the manufacturing process [4]. The most known example of this technology is Intel Optane.

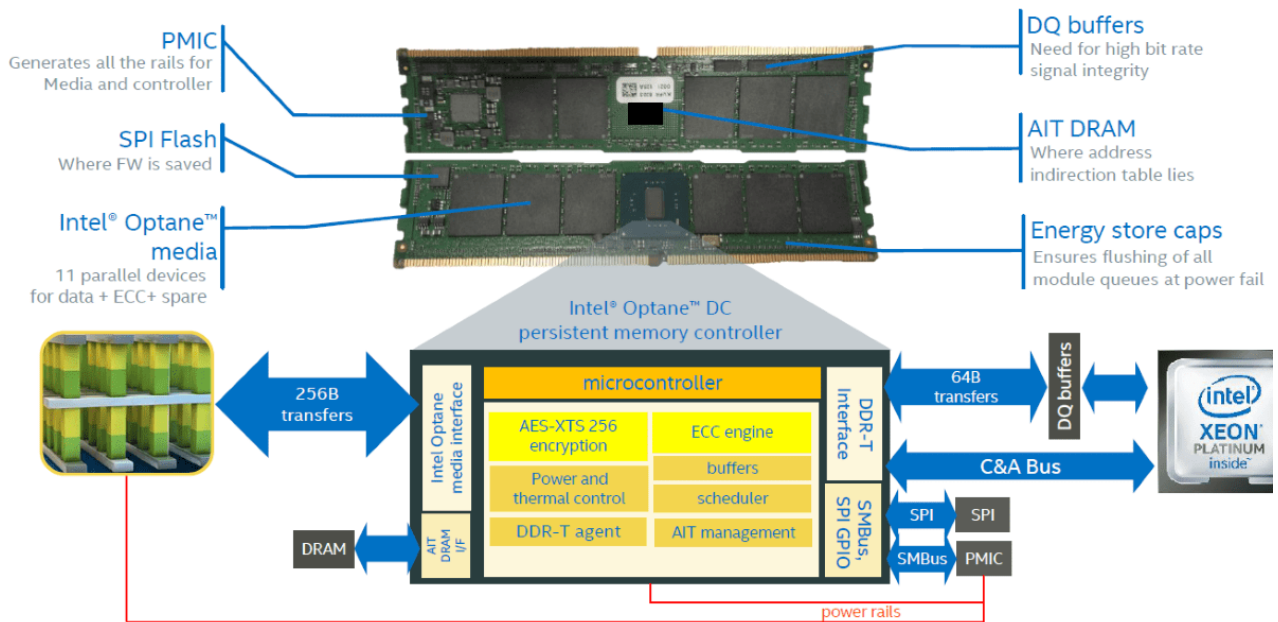


Figure 3 – Scheme of Intel Optane in DIMM form factor

1.1.2. NVRAM modes

Generally, there are two main modes in which NVRAM can operate. They differ in how system views NVRAM and which durability guaranties are being provided. Different parts of NVRAM could be configured in different modes. In this case, NVRAM is considered to be in mixed mode, but we can view them as an independent disk partitions, that allow different file systems to be configured on them.

1.1.2.1. Two level mode

This mode is also called memory mode and it is the simplest mode which does not require any changes in software being run on the machine or any additional prior knowledge. In this mode, NVRAM is visible to the system as ordinary random access memory and any present DRAM is hidden inside and used as cache. NVRAM itself in this mode becomes volatile and is used as cheaper but less effective system memory.

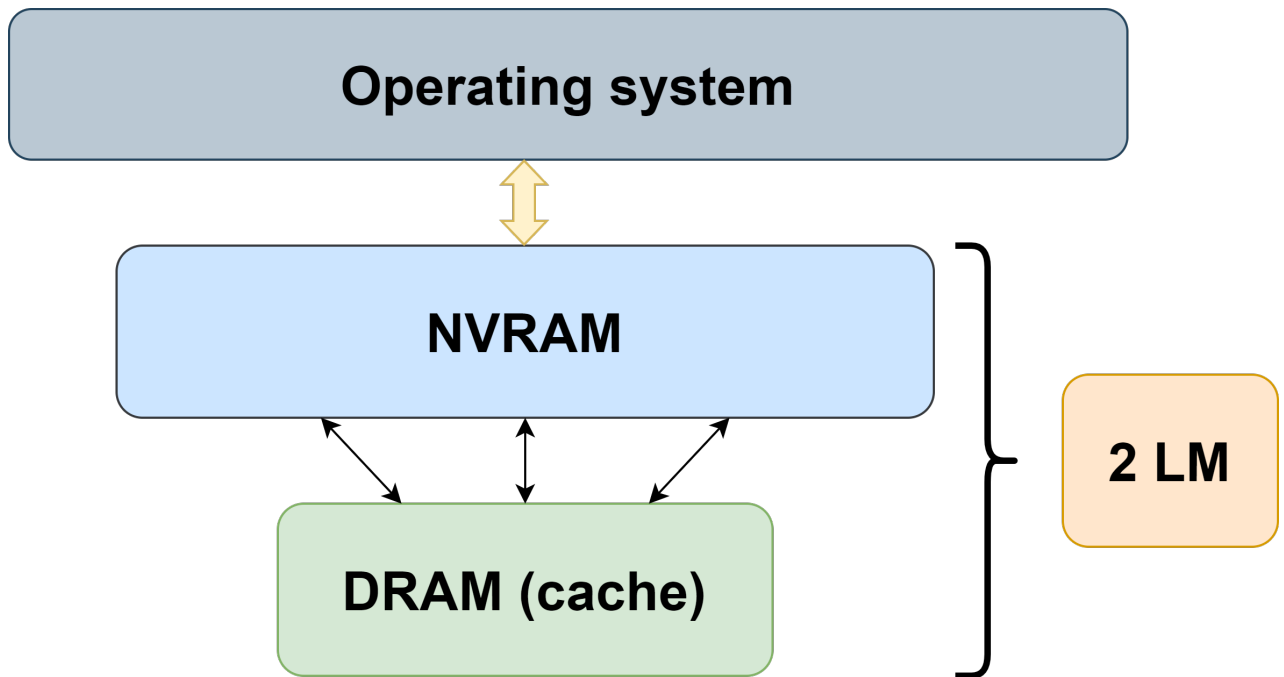


Figure 4 – Scheme of NVRAM configured in one level mode

Application that high data locality that are run in this mode can perform as fast as if they were run only on regular DRAM, however, now it is much more cheaper for them to reach such level of performance or in some case it was even impossible due to limited capacity of DRAM modules. Most notable examples of such application are in-memory databases like Redis.

1.1.2.2. One level mode

In contrast with two levels, one level offers opposite features:

- NVRAM is seen as independent memory device .
- NVRAM becomes non-volatile in this mode with regard to volatile caches mentioned before.

However this mode itself is divided in two sub-modes, depending on information how device itself was mounted: with direct access (DAX) or not.

If device is mounted without direct access it is visible as regular block devices with following properties:

- Block is a minimum addressing unit. So, even if you want to read a few bytes you have to read the whole block, which size is typically around 4 kilobytes.
- Operating system can cache this block in DRAM, so data accessed within one page will be loaded from NVRAM only once and after will be read from DRAM.

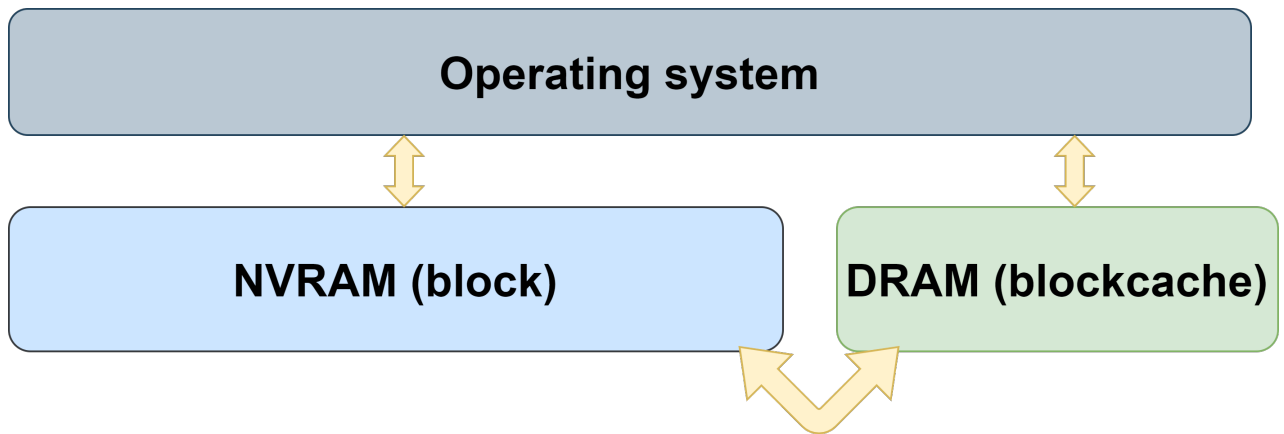


Figure 5 – Scheme of NVRAM configured in two level mode without DAX

For the end user, this sub-mode allows to view NVRAM just as essentially “fast solid state drive”. It also requires no changes in end users applications. It is considered particularly effective in workloads, that consist of sequential reads of big amounts of data, where it could perform up to seven times faster than classical storage systems.

However, this sub-mode doesn’t not allow to take advantage of ability to address individual bytes. On contrary, viewing NVRAM as a block device introduces an additional overhead. A lot of application rely on atomicity of block writes, which means they rely on the fact that either the whole block is written or not without any in-between states known as torn blocks. Traditional storage devices typically provide protection against torn sectors in hardware, using stored energy in capacitors to complete in-flight block writes, or perhaps in firmware. But with NVRAM we have atomicity guarantee only for one cache line, so extra steps are required in order to expose NVRAM as a block device.

This is done by a special data structure — block transaction table (BTT) (static layout shown on figure 6). BBT’s principle of work is described in more detail in [13], but in short it divides all available space in arenas of 512 gigabytes where global block address is mapped to internal block via indirection tables called BTT Maps and all writes actually happens in “allocating” manner in free blocks stored accessed via BTT Flog and only after they this blocks are redirected in BTT Maps.

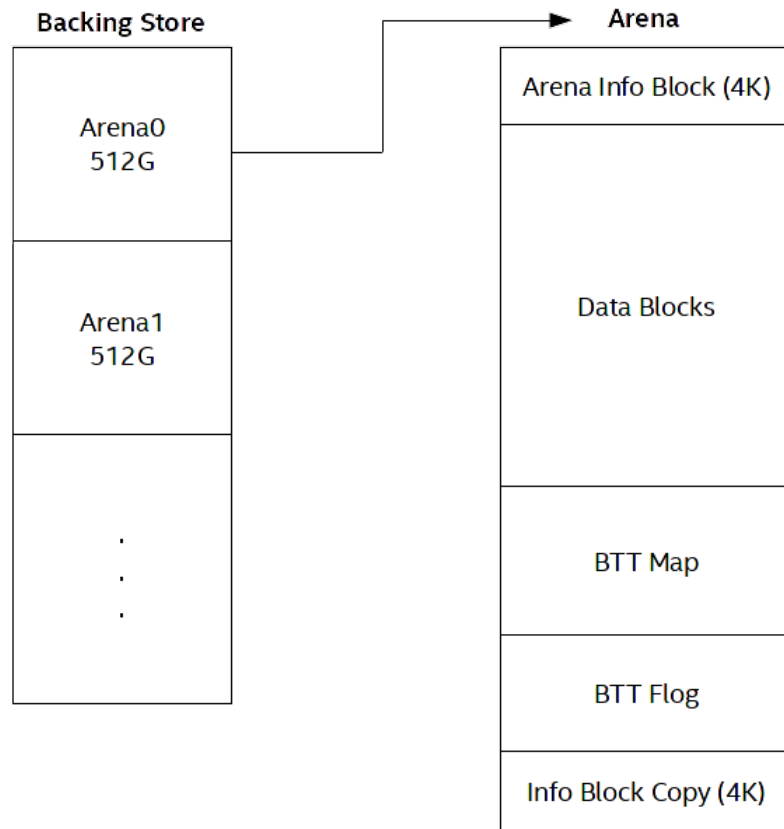


Figure 6 – BTT static layout

And only when NVRAM is configured in DAX sub-mode it truly opens all its capabilities providing the following properties:

- It is fully byte-addressable and supports fast random access
- Write atomicity is guaranteed on the cache-line (64 bytes) level,
- It is possible to map NVRAM directly to process memory, bypassing the kernel and its caches, which provides shortest code path to NVRAM allowing writing and reading directly from NVRAM. For instance on x86_64 it is possible to read from it via ordinary `MOV` instructions and persistently writing to it by flushing cache-line using specialised `CLFLUSHOPT`, `CLWB` or `PCOMMIT` instructions

However, to work with NVRAM as storage device it is required to use special file systems, that are aware of all nuances of NVRAM that differ it from classical non-volatile storages. Most notable being absence of sector write atomicity which most of the file systems rely on. Fortunately, some commonly used modern file systems such as `ext4` and `xfs` have already implemented support for NVRAM in their latest revisions. However, it is to be noted that NVRAM aware file systems

only guarantee consistency of its metadata is preserved without relying on sector write atomicity but no guaranties about writes of user data with sector atomicity is given, which application often may indirectly rely on. All of this specifics are schematically shown in figure 7:

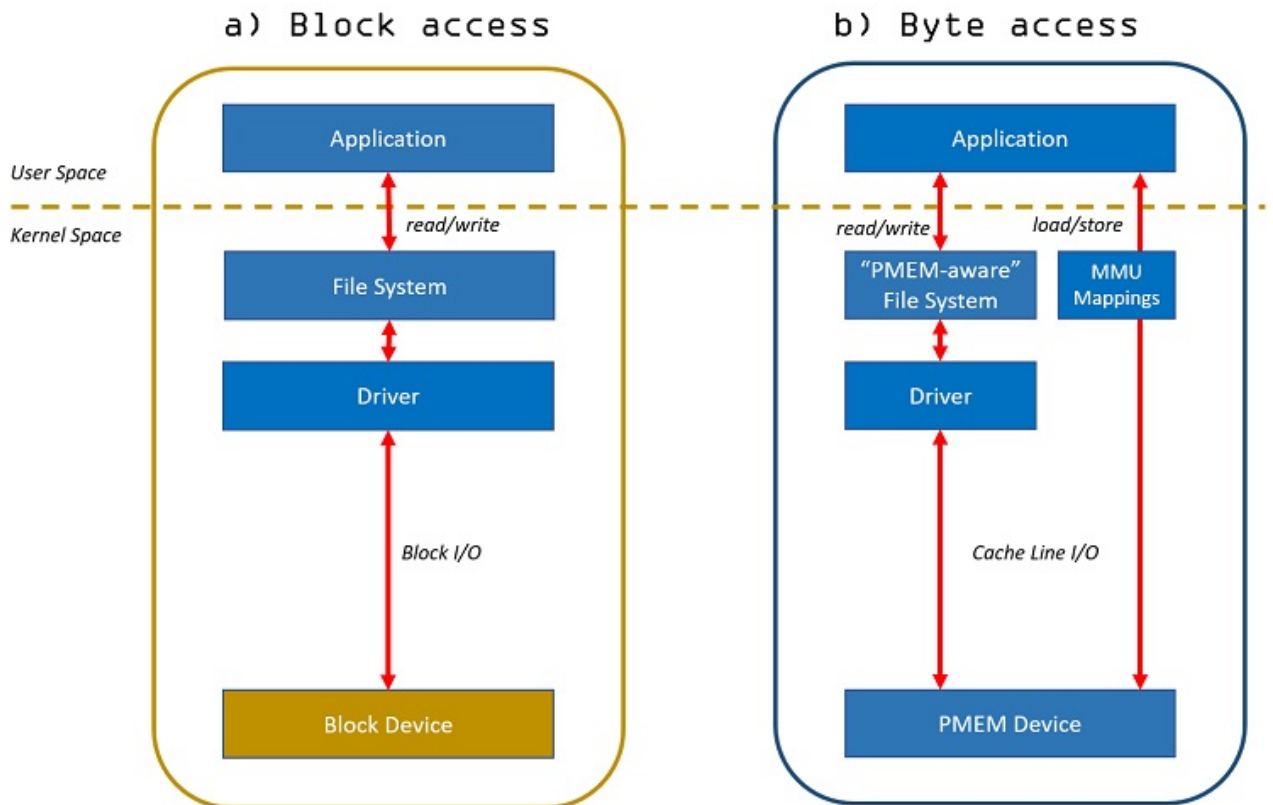


Figure 7 – Difference at the software level between block access (a) and direct access (b)

Last notable feature of the direct access is the ability to use NVRAM without non-volatility guaranty. In this case, there is no file system and NVRAM uses its internal caches are used more aggressively. This allows to view NVRAM as a additional slower RAM with more storage capacity. However this requires programmers to manually decide on which memory to allocate data allowing more precise control and implementing more specific logic of splitting data on “hot” and ”cold“ in any given case. Many operating systems show NVRAM in this configuration as additional NUMA node, which makes it easier to implement allocator for memory management.

1.2. Interaction with hardware

1.2.1. Device drivers

In today's world, electronic devices have become an integral part of our daily lives. These devices are composed of various hardware components, each with its own set of specifications and functionality. In order for software applications to interact with these hardware components, device drivers are essential. From a developer's perspective, device drivers serve as an interface between the hardware and the software, enabling software applications to communicate with specific hardware devices.

Without device drivers, software applications would not be able to interact with the hardware components of electronic devices, rendering them useless. For example, without a keyboard driver, software applications would not be able to receive input from the keyboard. Similarly, without a display driver, software applications would not be able to display output on the screen.

Device drivers are essential for creating software applications that can interact with specific hardware components. They provide a possibility for software applications to access the hardware, enabling the creation of complex and powerful applications. In addition, the device drivers ensure that software applications can run reliably and efficiently on electronic devices, providing a seamless user experience. The device drivers allow higher level programs to operate much more comprehensible abstraction such as block device, printer or network card without need to take in consideration which vendor produced particular device or what model does it have.

1.2.2. Operating system

However, having the device drivers is not enough to effectively develop any kind of sophisticated programs in reasonable time. Despite allowing to view lots of different hardware as generalized kinds such as a keyboard device or a network adaptor those abstractions are still very low-level and force programmers to take in consideration a lot of nuances related to specific hardware that is used in system.

In order to address this issue, another level of abstraction is introduced — an operating system. It provides a higher-level, more abstracted interface that simplifies the programming process and allows for the creation of complex software applications, schematically shown in figure 8.

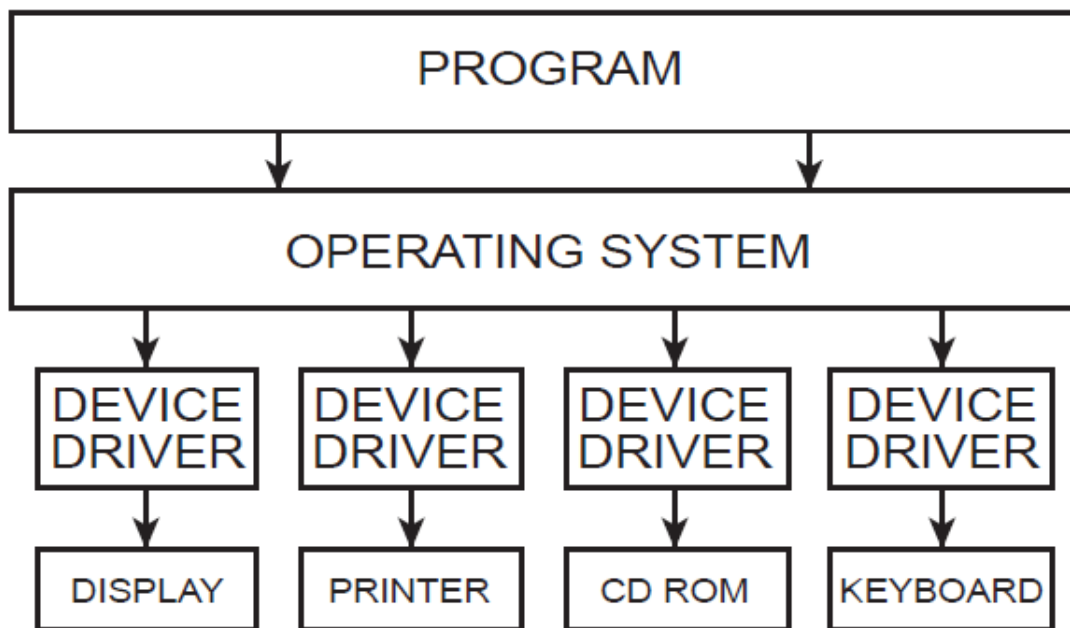


Figure 8 – Scheme of abstraction in modern system

An operating system provides a standardized API for accessing hardware resources, which simplifies programming by abstracting away the low-level details of device driver programming. This means that programmers can write code that is independent of the specific hardware details, making it easier to write code that is portable and can run on a wide range of devices.

However, this is not the only function of the operating system. This is only the part of its core component — kernel. The kernel provides most basic level control of all available hardware resources and performs various functions which allow execution of user programs:

- **Scheduling:** modern operating systems allow multiple programs to be run simultaneously on one system. However, numbers of programs drastically exceeds number of available cores on CPU. In order to make it seem like all this programs are run at the same time kernel allows each program to be executed for a fraction of time and then saves its state and starts executing different program.
- **Memory management:** all programs require some memory in order to represent their state and data, that they are working with. Memory on CPU also known as registers does not satisfy this needs, so systems use volatile and fast random access memory to do so. However, we can not give all available to one process because there are dozen of other processes waiting to be scheduled

and also require memory to be able to function. We also can't just divide all memory evenly between programs being executed because different programs require different amount of memory and also number of process changes over-time. This problems require advanced and dynamic memory management, which is performed by kernel.

- **Isolation**: most programs are written in assumption that they are the only ones being executed on the system and will continuously run until they are terminated. But as we see from previous points it is far from truth. However, this is a very practical model that allows a development of application without worrying about influence from other programs that are executed in the system. So, the kernel provides this “illusion” by carefully keeping track of what memory belong to which program and prevents programs from corrupting or even accessing each others data.
- **Inter process communication (IPC)**: despite each process existing in isolation from each other sometimes there is a necessity for them to communicate with each even if it is heavily limited, which kernel provides. It is done in a very careful manner in order not to break isolation too much, which will inevitably lead to data corruption. However, even legit actions can lead to errors if concurrence is not took into the consideration because modern CPUs allow truly parallel execution of commands which can lead to data races.
- **Security**: different actions have different influence on the system. For instance multiplying two integer numbers on CPU is generally considered a harmless action and does not have much effect on a system, but writing some data to storage devices could easily lead to data corruption or even data loss if not performed correctly. Even legit programs can perform actions which potentially can damage the system because of errors in their logic and that is not to mention programs which are intentionally designed to be malicious. In order to address these issues the kernel distinguishes which operations are allowed and which are not for particular programs introducing concepts as user, privileges, security rings and so on.
- **Networking**: not all programs are written in order to work within one system. Quite often they are working in conjunction with programs that are executed on other systems, but in order to do so they somehow need to communicate with each other. Devices such as network adapters allow to do so by transfer

data by wires or wordlessly using radio waves. However, in order to correctly send or receive data by these adaptors we need to decide what data belongs to each program, how to prioritise between them due to limited capacity of adapters internal memory and how much of system memory is used to buffer this data. All this is handled by kernel networking subsystems allowing programs to operate more simple abstractions such as Berkeley sockets [24].

- **Storage:** all functions mentioned above handle only volatile aspects of kernel functions. But it is also essential to store some information persistently which means this data must be kept intact even if there is no power on the system. Kernel itself has to be saved in such manner in order to allow system to function after it is restarted. Different storage devices such as hard drives and solid state drives allow to store data in a non-volatile manner, however they provide low-level functions of reading and writing sectors of flat data. Any complicated structures, transferring data between memory and storages, high level abstractions such a file systems that are used by most programs are provides by the kernel.

Those are not the only but just most notable functions that are provided by the kernel. An operating system itself performs many other functions apart from the basic system maintenance such as providing the stable public application programming interface (API) and an application binary interface (ABI) in order to keep source code and compiled binary programs portable from on system to another, handling real-time interruptions in order to respond to different input devices, giving ability to communications with output devices such as speakers and monitors in order to produce pictures and sounds that are comprehensible by humans and also providing user interface in order to be operated by users in form of either command-line interface where command are typed one by one from text input devices or graphical interface where some kind of graphical environment is used to allow users to interact with electronic devices through graphical icons and audio indicator which mostly uses input from a mouse rather then a keyboard.

1.2.3. System calls

Most programs that are written outside the development of operating system itself are written for the operating system itself rather than for any hardware components. In order to do anything apart from simply performing memory reads or

arithmetical calculations programs “ask” operating to do it for them with the help of system calls.

In an operating system, a system call is a mechanism that allows user-level processes to request services from the kernel or other privileged components of the system. Its role in the communication with the kernel is schematically shown on figure 9:

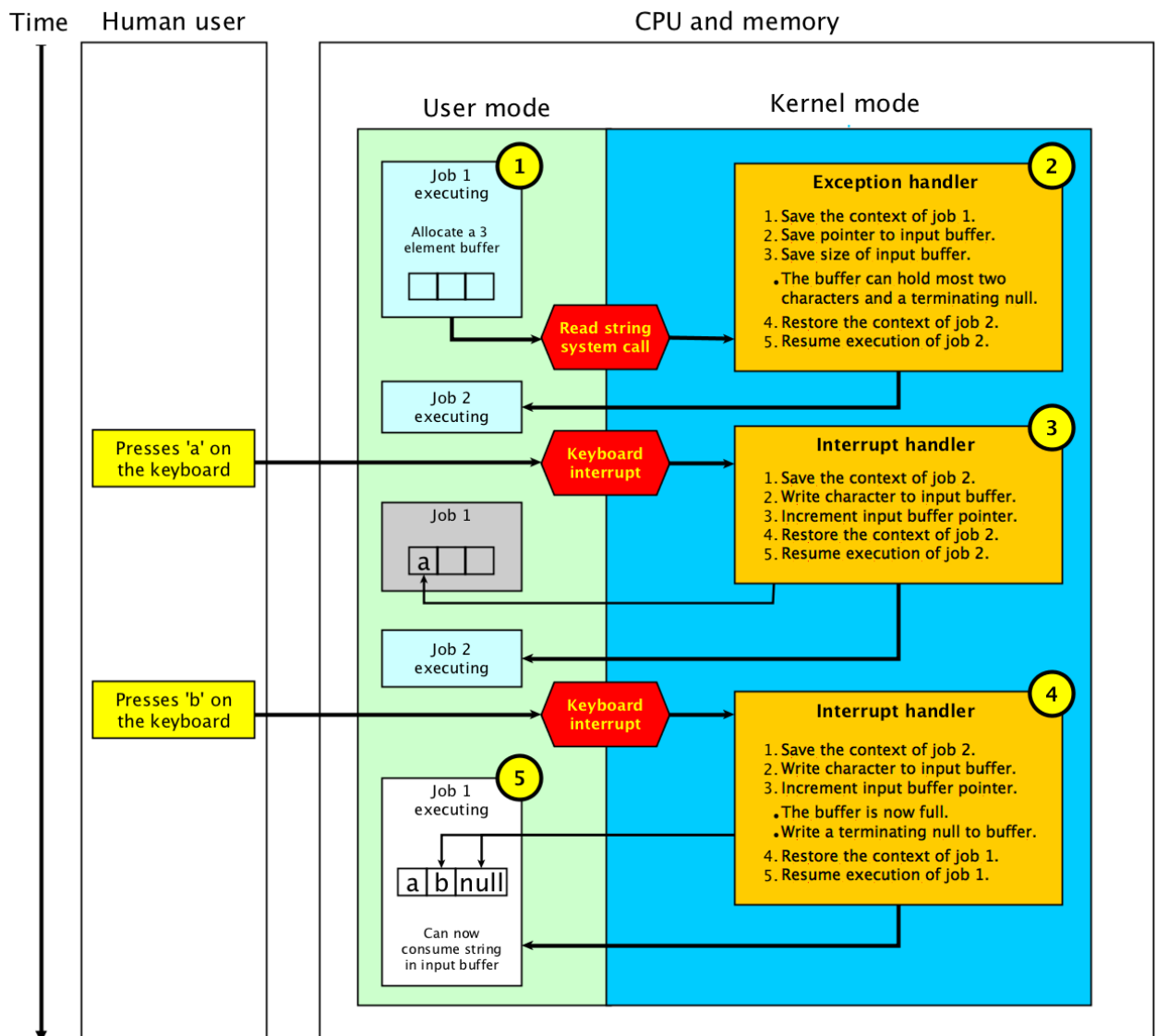


Figure 9 – The execution of processes in OS which interact with hardware

When a user-level process wants to perform an operation that requires privileged access, such as reading from or writing to a file, allocating memory, or creating a new process, it makes a system call. The system call transfers control from the user-level process to the kernel, which then performs the requested operation on behalf of the process.

System calls are typically implemented as functions provided by the operating system's application programming interface. The process invokes the system call by calling the appropriate function with the required parameters. The kernel then performs the requested operation and returns the result to the calling process.

Each system call has a unique identifier, known as a system call number or syscall number. The syscall number is used by the kernel to determine which system call the process is requesting.

Examples of commonly used system calls include:

- **open**: to open a file;
- **read**: to read data from a file;
- **write**: to write data to a file;
- **fork**: to create a new process;
- **exec**: to replace the current process with a new one;
- **exit**: to terminate the current process.

System calls are a fundamental part of an operating system's functionality and provide a way for user-level processes to access privileged services and resources in a controlled manner, which includes but not limits to:

- **Checking privileges**: Not every user in the system is allowed to do everything. For instance, some users may not be allowed to access networking as a security precaution, or changing vital system properties such as enabling overcommitting — allowing processes to allocate more memory than available in the hardware is given only to a user with high privileges.
- **Input validation**: Even if user is allowed to interact with some part of a system there can still be error in input information caused by mistake or malicious intent which can cause different kinds of corruptions in system. For instance, an user might try to perform a write operation to read-only device, write -1 or bytes to a hard drive or trying to pass pointer to memory that is not aligned properly.
- **Error handling**: Correct privileges and input parameters does not guaranty success of required operation. For example, there may be errors or underlying hardware such as corrupted data on hard drives and depending of nature of this corruption operation may be retried in some form of a recovery mode or maybe just device bus is overflowed at the moment and we should wait and

retry operation. In addition to recovering from errors inside the system there is also a need to propagate it to the caller in a way that the caller understands its severity and reacts accordingly. For instance, if a current process was just interrupted during system call execution by scheduler or real-time signal and this call returned this error indicated such interruption, user process can simply retry operation instead of propagating error further.

- **Scheduling optimisation:** Not all actions may take predictable time in order to be completed. For instance, reading an input from a keyboard or a network adapter takes indefinite amount of time because we do not know when user will press the button or new data will arrive from network. So, instead of actively waiting for data to be ready to be passed to program we tell scheduler to stop executing current process and replace it with some other process waiting to be executed. But in order to respond quickly we register handler to keyboard interruption, so when the data finally arrives we will be able to continue the execution of the original process right on. This situation is very similar to what we just described is illustrated on figure 9.

All this allows to view programs that are executed on the system as a stream of arithmetical operations and reads and writes from memory with occasional system call in-between which allows communication with other hardware.

1.2.4. File system

As mentioned above, one of the main functions of any operating system is the possibility to work with non-volatile memory also known as storage devices. Devices themselves provide very primitive API allowing to view them as contiguous array of data with some predefined minimum addressable unit. Without additional information there is no way to tell where some piece ends and where the next one begins, there is even no way to tell where the data is located to begin this or even tell if this part of device contains some useful information or just is vacant to put data on it.

In order to address this issue operating system provides a file system — a method used by operating systems to organize and store files and directories on a storage device, such as a hard disk drive or solid-state drive. It provides a hierarchical structure for organizing files and directories, as well as a set of rules for accessing and managing them. A file system covers several aspects in order to provide such functionality:

- **Space management:** It involves managing the available space on a storage device and allocating that space to files and directories. Without proper space management, it can be difficult to efficiently store and access files, and storage devices can become cluttered and disorganized. It includes several tasks:
 - **Block allocation:** File systems divide the storage device into fixed-size blocks and allocate one or more blocks to each file. This allows the file system to store files contiguously or non-contiguously.
 - **Free space management:** File systems keep track of the free space on the storage device, which is used to allocate space for new files.
 - **Fragmentation:** As files are created, modified, and deleted, the storage device can become fragmented, meaning there are alternating used and unused areas of various sizes. This may lead to a situation when there is no contiguous space for allocating file data, so it has to be stored in scattered pieces. For some devices such as hard drives this can significantly impact performance because even sequential data access in reality would result in random seeks along the device.
 - **Compression:** Some file systems support compression that can reduce the amount of space required to store files. This can be useful for storage devices with the limited capacity.
 - **Quotas:** File systems can also implement quotas that limit the amount of space that can be used by individual users or groups. This can help prevent users from consuming too much space on the storage device.
- **File organization:** File systems provide a way to organize files and directories into a hierarchical structure, typically using a tree-like structure. This is achieved by using directories. Directory is a special entity in file system which does not contain any user data but rather lists all files and other folders, that are present in it.
- **Multiple devices:** Most computers support connecting multiple storage devices to them, not mentioning that one device could be divided into multiple “subdevices”, called partitions. Different devices can be configured with different attributes or even different disk layouts entirely. However, user applications expect to be able to access all storage media. There are different techniques to do so. Windows does this with letter drives by assigning unique letters for each connected device. Linux takes approach of virtual file systems

[16] — a programming abstraction that allows applications to interact with the underlying file system in a standardized way, regardless of the specific file system implementation as if everything was placed in one file system.

- **Data integrity:** One of the key aspects of working with a non-volatile memory is inability to easily restart everything from scratch. Even after rebooting the system everything that you wrote will be present on the device. However, this is not safe in the presence of hardware errors or an unexpected power failure which may occur at any moment. This requires to keep file system consistent at any given time and carefully choosing algorithms and data structures depending on guarantees provided by device. Techniques such as checksums or even using error correction coding and journaling are often used to achieve this goal.
- **Access control:** File systems typically provide access control mechanisms to restrict access to files and directories based on user permissions. This can help protect sensitive data and ensure that only authorized users can access specific files.
- **Performance:** Apart from the usability and the consistency, it is obviously required from a file system to be as fast as possible. Different file systems may have different performance characteristics such as one is optimized for big files (for example, for a video media storage), others focus on huge amounts of small files (electronic mail services significantly rely on them). However there are common techniques that help to boost performance:
 - **Caching:** It is performed by saving most popular or recent pages with the data from a device to a fast volatile memory of the system. So, when this data is accessed next time for a read, we just return it from memory right away, and for a write we just modify page in memory and write it to disk later in background.
 - **Smart scheduling:** Another way to boost performance is to reorder input/output operations when possible in order to send requests to hardware more efficiently. This may involve batching read and write requests in order to better utilize throughput of a device bus, or re-ordering them in a more sequential manner which helps with hard disk drives, because they perform much more better when data is accessed sequentially due to construction specifics. Sometimes, it is possible to

eliminate all the communication with the device when write and read requests are in the same segments of data.

Apart from complex algorithms structures in systems volatile memory, all those aspects are implemented with the help of a special bookkeeping information associated with each entity within a file system — a metadata. The metadata contains various service information and can vary from file system to file system. However, it commonly contains the following information:

- **File size:** it can be both in bytes and blocks depending on what do you want to count;
- **Time:** it could be time when the file was created, modified, or last accessed;
- **Permissions and ownership:** information about who has permission to access and modify each file, as well as who owns the file;
- **File names:** information about the names of files by which users locate them in the file system;
- **File location and organization:** this metadata provides an information about the physical location of each file on the storage device, as well as how the files are organized within directories and subdirectories.

Most modern file systems store metadata and file data itself in separate places which allows operating system to change actual data layout, for example, during defragmentation, without blocking access to files for the running programs. In Unix-style file systems those metadata structures are called inodes (index nodes) because each inode has a unique index inode table, that describes structure of current file system. However, for example, NTFS has similar concept with unique fileID inside the master file table. Almost all file system operate with inode-like entities rather than filenames after locating file in some metadata table and they are the only unique identifier of file in whole file system because entities like symlinks in Linux allow different paths refer to same real file.

Last thing to mention about an application interaction with storage devices through operation system is durability. As we can infer from information presented above writing to a non-volatile storage does not often mean that the data is written in a non-volatile manner. Best that you could expect is that your data will be eventually placed on the storage media. One could suggest that we could perform writes that bypass any kernel caches or we could wait for data to be send to storage device. And

there are ways to do such as opening files on Linux with `O_DIRECT` flag, however this still does not guarantee that your data will be accessible after unexpected power failure. This happens because devices themselves also “cheat” by having internal caches of fast volatile memory which they use in order to speed up writes and reads and report back to kernel as soon as data is written to those caches. In order to do so they provide different commands in order to write data truly persistent such as ability to flush caches or make write bypassing caches entirely.

To allow an application that rely on data being truly and safely written to storages get such guarantees operating systems provide special functions that take in consideration all nuances motioned in previous paragraph: on Linux this function it is called `fsync`, on Windows it is `FlushFileBuffers`. Those functions ensure that all changes to the specific file are written to non-volatile memory, so, they could be visible even if system is unexpectedly rebooted right after this function finishes its execution. However, it is important to note that those writes are much slower compared to simply issuing write-like system call. Depending on particular operating system, file system, device and how often those sync are issued it could slow down performance up to 100 times.

1.3. Databases

One of most popular types of applications that heavily rely on file systems are databases. A database is a collection of data that is organised and stored in a structured way that allows for efficient retrieval and manipulation with the data. The data can be of various types, such as text, numbers, images, videos, and more. Despite a file system provides functions to store and organise data, its functionality is limited compared to features provided by a specific database that includes such things as:

- **Advanced data integrity:** not only databases may provide more complex errors correction codes but also maintain more advanced data constraints. For instance, with the help of database constraints it is possible for a phone book to guarantee that each person has at least one telephone number and each telephone number belongs only to one person.
- **Advanced data Security:** a database provides mechanisms to control access to data, such as authentication and authorization. This helps to protect sensitive data from unauthorized access.

- **Data scalability:** a database can handle very large volumes of data and support multiple users and applications accessing the data simultaneously. With the help of network communication database can store data on multiple systems when this amount exceeds capabilities of one system allowing itself to be easily scalable.
- **Advanced data retrieval:** it provides mechanisms to retrieve and analyze data in various ways, such as advanced queries that filter entities by flexible conditions that may even includes rules on data conjunctions, search entities by their contents such as words in documents or even fuzzy search that allows words to be misspelled, fast random access do data which is ordered by some arbitrary key and many more way that make it easier to extract insights and knowledge from the data.
- **Atomicity:** not every action in a file system is atomic, i.e., they either finish successfully or make no changes. It is possible for them to be partially done especially in the case of unexpected external errors. The simplest example is writing a data to a file: in the case of a power failure unpredictable subset off data will be persisted on storage.

There are different types of databases used depending on the specific data models and access patterns:

- **Relational Databases:** relational databases are the most widely used type of database. They store data in tables with columns and rows, and use SQL (Structured Query Language) to manage and manipulate the data [2]. Examples of relational databases include MySQL, Oracle, and Microsoft SQL Server.
- **NoSQL Databases:** NoSQL databases are non-relational databases that can store unstructured, semi-structured, and structured data. They are designed to handle big data and provide high scalability and availability. Examples of NoSQL databases include MongoDB, Cassandra, and Couchbase.
- **Object-oriented Databases:** Object-oriented databases store data in objects and classes, which are like templates for objects. This type of database is designed to work well with object-oriented programming languages like Java and C++. Examples of object-oriented databases include db4o and ObjectDB.
- **Hierarchical Databases:** Hierarchical databases store data in a tree-like structure, where each record has a parent and child relationship. This type

of database is commonly used in mainframe systems and is not widely used today.

- **Graph Databases:** Graph databases store data in nodes and edges, which represent the relationships between the data. This type of database is commonly used in social networks, recommendation engines, and other applications that deal with complex relationships. Examples of graph databases include Neo4j and OrientDB.

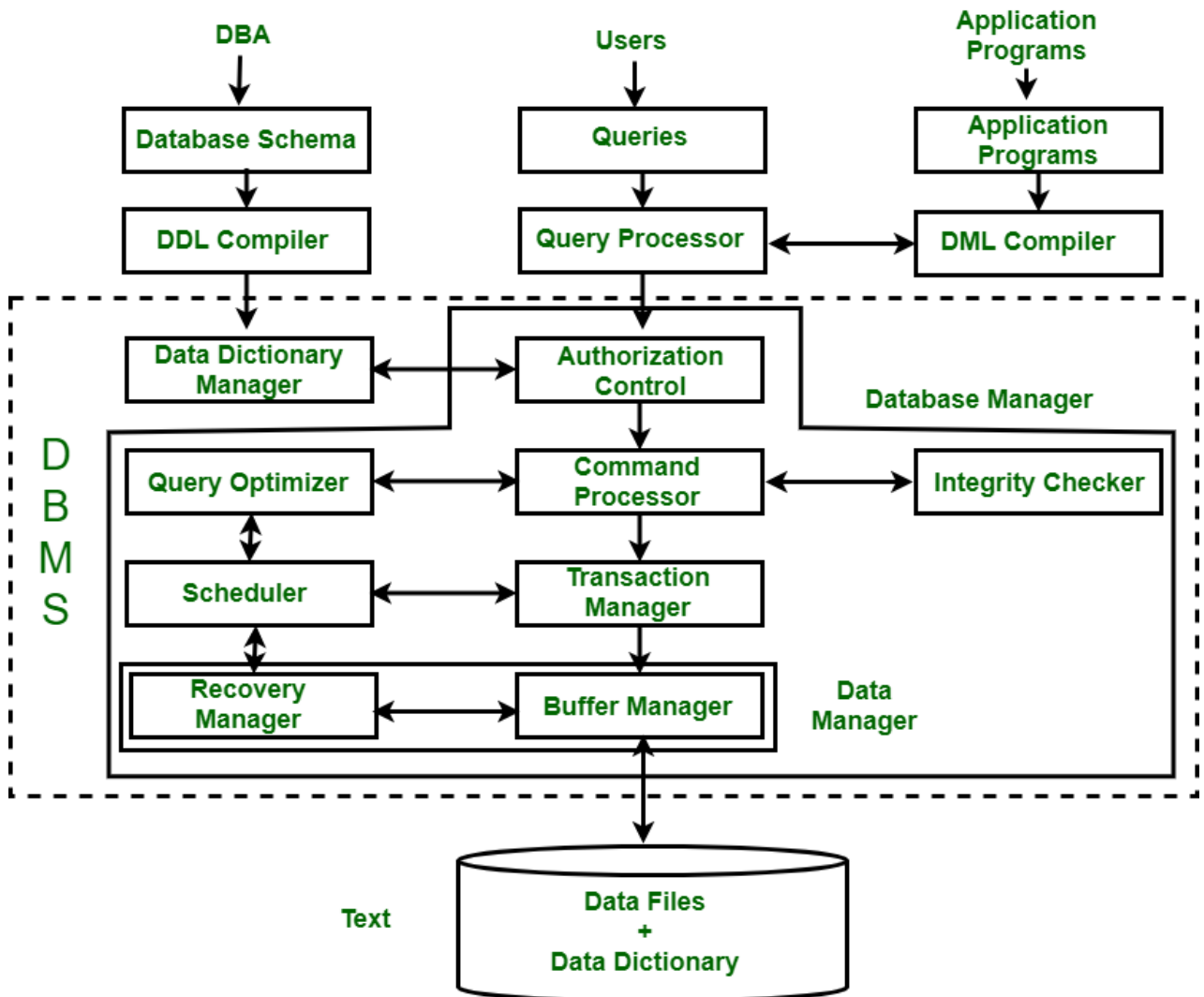


Figure 10 – Decomposition of database functions

However, a database itself is more of a concept or a public interface on properties of some applications. Real software that implements all those function is called database management system (DBMS). It handles many different aspects which compose databases and its responsibilities are schematically shown of figure 10. There are many different tasks that are handled, but for now I focus only on one — the transaction manager.

1.3.1. Transactions

As mentioned above one of the key features of databases is the advanced atomicity for many different actions it performs. However, this only one of the key properties of basic principal of interaction with database — transactions [5]. A transaction in a database is a logical unit of work that is performed on the data stored in the database. A transaction can be a single operation or a series of related operations that are performed as a single unit of work. Operations that must be executed in a specific order to ensure the integrity of the data. The four key properties of a transaction are commonly known as ACID:

- **Atomicity**: a transaction is atomic, which means that it is an all-or-nothing proposition. Either all the operations in the transaction are completed successfully, or none of them are. If any of the operations fail, the transaction is rolled back, and the database is restored to its previous state.
- **Consistency**: a transaction ensures that the database remains in a consistent state before and after the transaction is executed. This means that the transaction must maintain the integrity of the data, preserve any constraints, and ensure that the data is valid.
- **Isolation**: a transaction is isolated from other transactions, which means that the operations performed by one transaction are invisible to other transactions until the transaction is committed. This ensures that the data is not corrupted by concurrent access to the database.
- **Durability**: a transaction is durable, which means that once it is committed, the changes made by the transaction are permanent and survive any subsequent failures, such as a power loss or system crashes.

For sake of this work, we only focus on handling the durability aspect of transactions. This is done with the help of database journal. A database journal or a database log is a file that contains a chronological record of all the transactions that have occurred in a database system. Every time a transaction is performed in a database, such as adding, deleting, or updating a record, the details of the transaction are written to the database journal. This includes information such as the time of the transaction, the type of operation performed, the data involved, and any relevant metadata.

The database journal serves several important purposes. One of its primary functions is to provide a reliable record of all transactions that can be used to reconstruct the state of the database in the event of a failure. For example, if the system crashes, the database can be restored to its previous state by replaying the transactions in the journal that were not yet committed at the time of the failure. It is worth noting that the database log is updated in append-only manner and is often implemented via append-only files. This drastically simplifies recovery routines of database by providing an explicit linearization [3] of all concurrent transactions and also decreases likelihood of data corruption during writes to a persistent storage.

Thus, DBMS performance noticeably depends on the performance of synchronous writes provided by file system, because corruptions in other structures of database could be corrected by recovery routines with help of transaction log. It is not rare to see more relaxed durability guarantees where DBMS responds as soon as write returns from kernel rather than when file content is synced. This allows to significantly decrease response time of write queries but makes it possible to lose portion of last updates in case of a system failure.

Conclusions on Chapter 1

In this chapter, I performed an analysis of the subject area, to be more precise:

- Firstly, I presented a novel device — non-volatile random access memory by taking a look at its fundamental architecture, different working modes and features and limitations.
- Secondly, I presented basic principles of how programs interact with different devices installed in the system: they are executed on top operating system by merely asking it to perform some communication with device drives, which give very basic abstraction on top of specific hardware. In addition I took a closer look at storage aspect of operating system — a file system and described its functionality and basic principles and highlighted challenges of truly durable modifications to it.
- Lastly, I discussed database management systems — applications that heavily rely on file system capabilities and performance, especially in terms of synchronous writes to append only files.

All this leads to the main goal of this project — exploring possibility of using NVRAM to boost performance of synchronous writes to other non-volatile storage devices without changing source code of applications, that use them.

CHAPTER 2. GENERAL DESCRIPTION OF THE LIBRARY

2.1. Features and limitations

The main purpose of our work is to provide a NVRAM-based cache that is used to enhance the performance of synchronous writes to files. These are basic features of my library:

- No changes are required in the source code of application in order to take advantage of it.
- Both single-threaded and multi-threaded applications are supported.
- It tries to perform as much operations in user-space as possible in order to minimize expensive context switches to kernel.
- Specifying what files should be cached based on their extension and location.
- Customisable cache size, so the user can scale available NVRAM space for as many files as the user want
- Customisable cache location that eases maintenance of the device.

However, this library also imposes some limitations:

- Only append-only files are supported. Random writes require to mirror lots of functions performed by operating system and may require creating full scale file system from scratch, which is way out of scope of this project.
- Only one thread at a time can access files that are cached with our library. However in reality despite often having dedicated IO-threads applications rarely from different threads concurrently same files to say nothing of append-only files and even then they mostly have some explicit synchronisation in order to do so.
- Even within one thread file that is backed by cache could be opened only once before closing. If you want to access file again you must reopen it. Once again it is rare to see one file to be opened multiple times for different purposes. Most applications already close files before starting to use for different role.
- Library works only on Linux based systems with x86-64 architecture [10]. However, it is understandable how to extend its support to different architectures: this requires few platform-dependant switches in few places almost the same way Linux itself does it, but extending support to different operating systems requires additional consideration due to differences in how system calls are implemented there. Nevertheless, this does not affect the core prin-

cipals of how caching itself is performed, but rather complicates techniques used to allow no changes on applications that use this library.

2.2. Technologies used

2.2.1. libpmem

Unless we want to deal with low-level API provided by device drivers it is reasonable to use some kind of wrapper also known as library. Libpmem is a library for programming persistent memory devices, also known as Non-Volatile Memory (NVM). This library provides a set of APIs for developers to access and manage persistent memory devices.

The library supports a range of persistent memory devices, including Intel Optane DC Persistent Memory, which is a new class of memory that combines the speed of DRAM with the persistence of NVM. It also supports other types of NVM, such as NVDIMMs, SSDs with integrated NVM, and PCI Express-based NVM [14].

The goal of libpmem is to provide a programming interface that allows developers to treat persistent memory devices as if they were volatile memory, while still retaining the durability and persistence benefits of non-volatile memory. It allows applications to directly read and write persistent memory, bypassing the file system and operating system buffering if it was configured in an appropriate mode. It provides API very similar to one that Linux provides for working with memory mapped files. Consider examples provided in code snippets in listings A.1 and A.2:

a) Code on listing A.1 does the following:

- 1) opens a file located at `PATH` or creates it if does not exists;
- 2) ensures that it has enough space for `LEN` bytes of data;
- 3) maps file contents to special memory pages returning pointer `ptr` and closes the original file descriptor;
- 4) writes string to `ptr`;
- 5) ensures that the written data is persisted by explicitly flushing changes made to the in-core copy of a file;
- 6) deletes a mapping associated with `ptr`.

It also performs some basic error handling but for sake for simplicity I skip it.

b) Code on listing A.2 does following:

- 1) creates a mapping via pointer `ptr` for a file located at `PATH` ensuring that such file exists and has enough space for `LEN` bytes of data;
- 2) Ensures that such file was located on NVRAM;

- 3) writes a string to `ptr`;
- 4) ensures that the written data is persisted;
- 5) deletes a mapping associated with `ptr`.

Once again basic error handling is performed but we do not focus on that.

From this descriptions, it easy to see that both programs perform almost identical tasks and even almost identical steps that vary how actions are grouped in one function call. This is not a coincidence, `libpmem` was designed to be similar to `mmap` [21] to extent. The code which worked with pointers provided by `mmap` should continue to work with a pointer provided by functions from `libpmem`. In a nutshell, `libpmem` provides low-level APIs that allow developers to allocate, deallocate, and manage memory on persistent memory devices, as well as read and write data to and from these NVM devices. This API provides tools to ensure durability aspect of your program. However, it does not pose a transactional property, so, the atomicity and consistency aspects must be taken care of solely by programmer:

- **`pmem_map_file`**: creates read/write mapping for a given file, optionally ensuring that it exists and resizing it. It allows creating such a mapping either on NVRAM or classical storage device and reports where the mapping took place. The mapping itself is performed internally by `mmap`, but it also takes extra steps to make large page mappings more likely which speed up MMU routines that locate physical page by virtual memory address.
- **`pmem_unmap`**: a function similar to `munmap` that deletes mappings created `pmem_map_file` and causes further references to addresses within the range to generate invalid memory reference.
- **`pmem_flush/pmем_drain`**: these functions allow to flush any processor caches and device caches respectively. These functions are suitable only for memory returned by `pmem_map_file` that was mapped to actual NVRAM. If you want to perform both actions there is convenient wrapper `pmem_persist`. Main difference of this functions compared to `msync` is that these functions will, if possible, perform the flush directly from user space, without calling into the OS. For example on the Intel platform it could be achieved by using instructions like `CLWB` and `CLFLUSHOPT`. `Libpmem` checks the platform capabilities on start-up and chooses the best instructions

for each operation it supports.

These functions satisfy all the basic requirements in order to create more complex programs such as transactional object store as in `libpmemobj` or transactional pmem-resident log file as in `libpmemlog`. However, in order to simplify efficient copy of data to NVRAM this library also provides functions that are similar to *libc* functions `memcpy`, `memset`, and `memmove`, that are called similarly with prefix `pmem_` and optionally provide properties of `pmem_flush/pmем_drain`. This also done by using platform specific instructions, like non-temporal store instructions on Intel which bypass the processor caches.

2.2.2. `syscall_intercept`

Syscall interception is a technique used in operating system design and development to intercept and modify system calls made by processes running on the system. The idea is to intercept system calls before they are executed by the operating system kernel and to provide additional functionality or security checks.

The `syscall_intercept` library is a popular open-source implementation for a syscall interception. It allows developers to intercept system calls on Linux systems with `x86_64` architecture and to modify their behavior in a variety of ways. Developers can intercept system calls and redirect them to custom handlers, modify system call arguments, and inject custom code into the execution path of the system call. This allows developers to add custom functionality or security checks to the system call [22].

API of this library mainly provide two functions:

- **`intercept_hook_point`**: a global function pointer to interception callback, that must assigned in function with the constructor attribute in order to be initialised at start. This function must accept a number of syscall as its first argument and six arguments of type `long` for possible syscall arguments. It also takes argument with the pointer for result because return value of function itself signalises if user decided to intercept syscall or it should be passed to kernel.
- **`syscall_no_intercept`**: a convenience function that takes syscall number and an arbitrary number of syscall arguments that allows to issue a system call that will not be intercepted. It is similar to standard *libc* function `syscall`

however return value should be handled differently.

One common use case for syscall interception is in the development of security software, such as antivirus or intrusion detection systems. By intercepting system calls, security software can monitor and analyze the behavior of processes on the system and detect malicious activity.

Another use case for a syscall interception is in the development of debugging tools. Developers can use syscall interception to track system call usage and diagnose issues with applications.

Under the hood interception is done by hotpatching *libc* binary that is already loaded to the process memory. The library disassembles the text segment of the *libc* loaded into the memory space of the process it is initialized in. It locates all syscall instructions, and replaces each of them with a jump to a unique address. Since the syscall instruction of the x86_64 ISA occupies only two bytes, the method involves locating other bytes close to the syscall suitable for overwriting. The destination of the jump (unique for each syscall) is a small routine that accomplishes the following tasks:

- a) Optionally executes any instruction that originally preceded the syscall instruction, and was overwritten to make space for the jump instruction.
- b) Saves the current state of all registers to the stack.
- c) Translates the arguments (in the registers) from the Linux x86_64 syscall calling convention to the C ABI's calling convention used on x86_64.
- d) Calls a function written in C (which in turn calls the callback supplied by the library user).
- e) Loads the values from the stack back into the registers.
- f) Jumps back to *libc*, to the instruction following the overwritten part .

To be precise, an original binary application could be modified in two different ways. When `syscall` and surrounding instructions can be overwritten it just places `jump` to special routine as shown on figure 11:

Before:	After:
db2a0 <__open>:	db2b0 <__open>:
db2aa: mov \$2, %eax	/-db2aa: jmp e0000
db2af: syscall	
db2b1: cmp \$-4095, %rax	db2b1: cmp \$-4095, %rax ---\
db2b7: jae db2ea	db2b7: jae db2ea
db2b9: retq	db2b9: retq
	...
	...
	_...
	e0000: mov \$2, \$eax
	...
	e0100: call implementation /
	... /
	e0200: jmp db2aa _____/

Figure 11 – Simple hotpatching

However, sometimes the instructions directly preceding or following the `syscall` instruction can not be overwritten, leaving only the two bytes of the `syscall` instruction for patching. In this case hotpatching library looks for a place for the trampoline jump in the padding found to the end of each routine. Since the start of all routines is aligned to 16 bytes, often there is a padding space between the end of a symbol, and the start of the next symbol. On figure 12 that illustrates hotpatching using a trampoline jump below, this padding is filled with seven byte long `nop` instruction, so the next symbol can start at the address 3f410:



Figure 12 – Hotpatching using a trampoline jump

2.3. Library implementation

Now, I take a closer look on how this library is providing all the features, that I mentioned in the beginning.

2.3.1. General design

To comply with the original requirement of demanding no changes in source code of original applications this library bases it on intercepting system calls that original application issues in order to interact with a file system. It performs the following steps:

- a) Intercepts syscall from original applications.
- b) Analyses this syscall and if it is not related to file system passes it to kernel, otherwise, proceed further.
- c) Analyses if syscall is related to files, that are cached via this library. If not once again passes syscall to kernel, otherwise, proceed further.
- d) Mimics effects of original syscall and returns control back to original application.

Already, it possible to see where reduced number of syscalls comes from: due to optimisation in libpmem that allow interaction with NVRAM directly from user-space it is possible that the whole interception will be performed in user-space too.

2.3.2. Parameters passing

In order to control the behavior of our library we need to somehow pass customisable parameters to it. For a regular application it is done via command-line arguments, however, because we cannot change source of original application any attempt to use this technique would mostly lead to errors in parsing routines of an original application or even undesired changes in its behaviour.

So, in order to solve this problem we use a different tool — environment variables [25]. An environment variable is a dynamic value that can be set and accessed by software running on a computer system. These variables are essentially key-value pairs that provide a means for software applications and operating systems to communicate with each other and share information. They do not change the command-line that is consumed by the program, however, often can change program's behaviour as shown in listing 1:

Listing 1 – Invoking date utility with different TZ environment variable

```
dogzik@DESKTOP-LEV:~$ TZ='America/Los_Angeles' date --rfc-email
Sun, 14 May 2023 09:32:07 -0700
```

```
dogzik@DESKTOP-LEV:~$ TZ='Europe/London' date --rfc-email
Sun, 14 May 2023 17:32:23 +0100
```

Our library takes its arguments from the predefined environment variables that are prefixed with string `SUFFIX_CACHE_` in order to avoid clashes with other environment variables that are used by the system and/or applications. Such variables include the following:

- **SUFFIX_CACHE_EXTENSIONS_TO_FOLLOW**: list of file extension used to filter which files must be cached by this library and which not.
- **SUFFIX_CACHE_SIZE**: the number that tells how much data is used for each files cache in bytes.
- **SUFFIX_CACHE_PATH**: the path to a location where cached would be placed. This path must reference folder on NVRAM in order for library to work.

- **SUFFIX_LOAD_FACTOR**: the threshold for cache's load factor after which automated flushing to underlying device starts which will be discussed later.

2.3.3. Intersection subtleties

The intersection in this library is made with the help of library `syscall_intercept`. However, I have to address two main problems:

- **multi-threading**: we need to be able correctly intercept syscall that are made concurrently from different threads of the application.
- **reentrancy**: sometimes we need to perform some syscall inside the handler while already intercepting another syscall. Although library provides possibility to issue syscalls without interception, this API is very low-level and usage of convenient wrappers from *libc* is much more desirable.

In order to overcome these challenges I track current state of interception with a special boolean flag — `in_hook`. Initially it is set to `false`, and during the interception it is set to `true` before performing any actions. Main trick is that if variable is declared as `static`. In C++, a static variable declared inside a function is a variable that retains its value between function calls. This means that the variable is initialized only once, the first time the function is called, and retains its value across subsequent calls [8]. Static variables in functions can be useful when you want to retain some state information between function calls. For example, you might want to keep track of the number of times a function has been called as shown in listing 2 below:

Listing 2 – Static variable in C++

```
void myFunction() {
    static std::size_t callCount = 0;
    callCount++;
    std::cout << "This function has been called " << callCount <<
        " times." << std::endl;
}
```

So, when we start executing our intercept function: firstly, we check this `in_hook` variable, and if it is set to `true` we return from the function immediately indicating that syscall should be passed to kernel.

However, this covers only the reentrancy aspect of our problems. In multi-threaded applications we could encounter syscall from different threads during the

interception one from current which may lead to situation where we pass `syscall` from a different thread to the kernel because we encounter `in_hook` in `true` despite that in reality we need to intercept it in order to function correctly. Simplest solution in this case would be protecting `in_hook` variable by some kind of lock like for example `std::mutex`. This will ensure correctness of our interception, however, it will introduce unnecessary dependencies and the waiting time during the execution, that negatively affects the performance of the original application.

Looking closely, I saw that a decision about the interception is really independent in each thread and we only care about reentrancy within one thread unless we explicitly perform additional actions in another thread during interception. In this case, we have the full control of a source code and knowledge of the internal logic and can perform additional logic for this situation. However, in order to achieve convenience in common case we can use different tool — `thread_local`. In C++, `thread_local` is a storage class specifier that indicates that a variable is local to a thread. This means that each thread has its own copy of the variable, and changes made to the variable by one thread will not affect the value of this variable in another thread. They are initialized once per thread, at the time the thread is created, and when a thread is terminated, such variables are destroyed [9]. So, we mark our variable with this specifier and final code is similar to code below:

Listing 3 – Final interception hook

```

1  int hook(long syscall_number , long arg0 , long arg1 , long arg2 ,
      long arg3 , long arg4 , long arg5 , long *result) {
2      static thread_local in_hook = false;
3      if (in_hook) {
4          return PASS_TO_KERNEL;
5      }
6      in_hook = true;
7      if (need_intercept(...)) {
8          *result = do_intercept(...);
9      } else {
10         in_hook = false;
11         return PASS_TO_KERNEL;
12     }
13     in_hook = false;
14     return INTERCEPTED;

```

Now, as shown on listing 3, each thread has its own state of interception. So, when each thread encounters its first call to interception function it will initialise `in_hook` variable to `false` and proceed with interception passing check

shown on line 3. After this if thread decided not to proceed further by check evaluating `need_intercept` to false on line 7 it “unlocks” the hook on line 10, so the following invocation of the hook in this thread is able to pass a check on line 3 too. Otherwise, if this thread proceeds further with the interception by calling `do_intercept` on line 8 all syscalls within that function will be passed to kernel because each check on line 3 will fail and lead to return on line 4, and after this we once again “unlock” the hook on line 13 to allow next interceptions to proceed the same way.

2.3.4. Initial interception

As shown previously, our interception hook is invoked on every syscall issued from original application, however, in my library I am only interested in actions that are related to a file system. So, it reasonable to perform initial filtering of intercepted syscall by immediately passing to kernel syscalls that could no be related to file system. As shown in chapter 1, every syscall has a unique number by which they are identified in system and these numbers are available in the system header file named `sys/syscall.h` by `SYS_*` constants. This allows to implement function `need_intercept` on line 7 in listing 3 similar to the following way:

Listing 4 – Simlified implemetation of `neen_intercept` function

```
bool neen_intercept(long syscall_number) noexcept {
    switch (syscall_number) {
        case SYS_open:
        case SYS_close:
        case SYS_fsync:
        case SYS_write:
        case SYS_read:
        case SYS_lseek:
            return true;
        default:
            return false;
    }
}
```

2.3.5. Intercepted syscalls

Now, that we filtered all uninteresting syscalls, we can focus on what we really want to intercept — file system related ones. Generally, they could be divided in several logical groups:

- **Open syscalls:** these syscalls produce a new file descriptor by creating new handle to a file in the memory of current process;
- **Close syscalls:** these syscalls do the opposite and invalidate previously created file descriptor by destroying handle that it refers to;
- **Sync syscalls:** these syscalls are used to explicitly persists changes to file, that might be stuck in different buffers as discussed in chapter 1;
- **Write syscalls:** as name indicates this syscalls modify files by adding new data to them or replacing the existing;
- **Read syscalls:** as name indicates this syscall fetches data from files. There is also a guarantee that all data produced by writes that precede current read will be visible to it even if this data is stuck in some buffer;
- **Seek syscalls:** these syscalls change position of current pointer inside handler that is referenced by a file descriptor.

I discuss each group of syscalls separately and subsequently present the architecture of the interception runtime.

2.3.5.1. Open syscalls

Mainly this syscalls are present by the following four:

- **open:** This system call is used to open a file or create a new file if it doesn't exist. It takes a filename and a set of flags as arguments, and returns a file descriptor that represents the opened file. The file descriptor is an integer value that can be used in subsequent operations, such as reading from or writing to the file. The “open” system call operates relative to the current working directory of the process.
- **creat:** This system call is equivalent to calling `open` with flags equal to `O_CREAT|O_WRONLY|O_TRUNC`.
- **openat:** This system call is similar to `open`, but it provides a more flexible way of opening files by allowing the specification of a directory file descriptor as a starting point for the path resolution.
- **openat2:** This syscalls was recently introduced as an attempt to create a universal way to open and create files with the potential for creating future extension without creating new syscalls as it happened with `openat`. However, it is not widely supported as of today and would not supported in my library for now.

Basing on this information we could focus on implementing only `openat`. The interception of others is done by just calling the interceptor of `openat` with appropriate parameters. Therefore, I discuss only `openat`.

After receiving a request to open a file, we do the following:

- a) we analyse original syscall parameters in order to determine if we need cache each particular file or not;
- b) we check that the file is either write-only or read-only because we do not support mixed workloads for now;
- c) for write only files we check that they are opened as append-only files;
- d) If everything checks out we continue to process the file;

The first step is done by checking `pathname` argument of original syscall. Unfortunately, here we once again encounter challenges related to multi-threaded nature of original application. In order to filter files via their path we convert some of library arguments to convenient filtering structure, however, this conversion requires some parsing and memory allocations and it would be inefficient to do so every time we want an open file. In order to optimise this, it is reasonable to create a filtering function with a static instance of filter in it. However, what happens if another thread calls this function before the initialisation is complete? We could also make this with a `tread_local` variable as we previously did. However, in this case, every thread has its own copy of this filter and perform its initialisation despite there is no need to do so. Fortunately, for us, the C++ standard states that if multiple threads attempt to initialize the same static local variable concurrently, the initialization occurs exactly once. Moreover, the usual implementations of this feature use variants of the double-checked locking pattern, which reduces the runtime overhead for already-initialized local statics to a single non-atomic boolean comparison.

The second step is a simple check that ensures that `flags` arguments does not include `O_RDONLY` flag.

The third step also simply checks `flags` that ensures that they include either `O_TRUNC` to create an empty file or `O_APPEND` that indicates that all writes to the existing file is append-only.

If any of those checks fails we just call the original `openat` syscall and return its result from the hook. Otherwise, we proceed with our interception. For write-only files it goes as following:

- a) we open an actual file in order to obtain file descriptor to return to the application;
- b) we create an internal handler that is used for the operation with this file;
- c) we associate a file descriptor with an internal handler for future interceptions and return the file descriptor to hook.

The first step is fairly simple call to the original `openat` with some basic error handling and does not require many comments. The second step is discussed later when I describe the handler. However, the third step requires an additional discussion.

In order to create such association we need some map-like structure. And right from the start we must take into consideration multi-threaded nature of original application which might lead to situation where several such associations are created concurrently. There two main options for this:

- Knowing that a file descriptor is just a number, it is reasonable to use a simple array. In this case, a different request always accesses different elements of this array due to the uniqueness of file descriptors, and knowing that only one thread access each particular cached file at a time we conclude that there would be no concurrency related problems, assuming that every elements hold something like atomic pointer.
- We use ordinary map structure like `std::unordered_map`. In this case concurrent requests become an issue and simplest solution would be protecting this map by a lock.

Despite the first option requiring no locking it requires preallocating array big enough to use any possible file descriptor as an index. In Linux, the maximum number of file descriptor is a configurable parameter, that could be changed at any time and file descriptor itself having type `int` may lead to allocating too much memory only for library's internal structures. Adding to this possible multiple instances of library being run simultaneously make this option much less favourable as it initially seemed.

In addition, the standard C++ map collections are based on nodes which guarantees us a pointer and reference stability — references and pointers to either key or data stored in the container are only invalidated by erasing that element. So, even

rehashing, which might happen after creating new association, does not invalidate a pointer and references. This allows us to use more efficient read/writes locks, that allow multiple readers to access protected code/data [7], because modification of map content itself happens only when associations are created and removed which happens much less often compared to access to files. It is also worth noting that the same effect could be achieved by storing pointers to handlers allocated somewhere on heap rather than inside map itself if we decided to use different map implementation, which for example may support more fine-graded locking or even be lock-free.

Read-only files follow similar logic, with one notable difference — we do not create mappings for them. In order to know why we need to take a look at their handling:

- a) We open a file descriptor for them, however, we open them with `O_WRONLY` and `O_APPEND` flags.
- b) We create a handler for this file using a file descriptor from first step and forcefully flush data to the original file.
- c) We close the file descriptor from the first step.
- d) We open the file again exactly how it was originally asked and return this from the hook.

In this case, I ensure files consistency in order to be later accessed as usual by the kernel, which may use more sophisticated caching techniques for the read access. Our primary goal is to boost synchronous writes, so we do not introduce any unnecessary complications for reads.

2.3.5.2. Close syscalls

There is only one syscall, that closes files — `close`. It takes one argument — a file descriptor, so its interception follows a simple algorithm presented below:

- a) locate a handler for given file descriptor in our map with a **read** lock;
- b) if we find nothing, immediately call `close` on a given file descriptor and return from the hook;
- c) we use a handler to forcefully flush data to the original file;
- d) we erase an association for the given file descriptor from the map with a **write** lock;
- e) call `close` on the given file descriptor and return from the hook.

Close is one of few syscalls that requires an exclusive access to shared state, so, we minimise the time that it spends in critical section with this level of access.

2.3.5.3. Sync syscalls

There are two main syscalls to ensuring the file durability — `fsync` and `fdatasync`. The first one, as previously explained, ensures that all modified in-core data of file is persisted on storage device, the second implies less strict guarantee — it ensures that only the information related to user data inside file is durably saved onto a storage device, however, some metadata such as the last access or the modification time might not be correct if a power crash happens right after this syscall [12].

However, on classical storage devices it takes only one write to save all metadata, so, in the case of append-only files that constantly change their size there is not much difference in performance. This is also the reason why interception of this syscall is done identical in our library and is done as following:

- a) locate the handler for a given file descriptor in our map with the **read** lock taken;
- b) if we found something, we immediately return because those operations are no-op due to the nature of our handler, that will be presented later
- c) otherwise, we just pass the requested syscall directly to the kernel.

2.3.5.4. Write syscalls

Generally there two type of write syscall that we want to distinguish:

- **Positioned**: a syscall is prefixed with letter `p` like `pwrite` and takes explicit offset from which write starts. It requires files, that are capable of seeking, and do not change internal offset stored in file handler that is referenced by the file descriptor;
- **Unpositioned**: this syscall on contrary uses starts writing from an internal offset, that is stored inside OS file handler. After write is completed this internal offset is incremented by the number of bytes written. The simplest example would `write` syscall.

Because we support only append-only files we efficiently can intercept only the second group of these syscalls. However, we cannot know beforehand which syscalls would be used especially if the file was opened with `O_TRUNC` flags, so

it is possible after several appends positioned write was called. In order to provide the best effort support we intercept such writes to files that might be cached in the following way:

- a) locate a handler for the given file descriptor in our map with the **read** lock taken;
- b) if we find nothing immediately call original syscall on the given file descriptor and return from the hook;
- c) we use the handler to forcefully flush data to original file;
- d) we erase the association from the map for the given file descriptor with the **write** lock taken;
- e) we call the original syscall on the given file descriptor and return from the hook.

However, when we intercept the positioned write syscall we finally get to the original task of our library. Mainly there are two types of such syscalls:

- **Scalar**: these syscalls take one contiguous buffer of data of input and write out some prefix of it to file referenced by given file descriptor.
- **Vector**: these syscalls take several buffers of data that might not be contiguous in memory but considered to be so logically, and write them out, guaranteeing that buffers are processed in array order, so each buffer data is written to the disk when all previous buffer are written out.

It is easy to see, that the first type can be expressed with first and we do so and focus only vector writes to file. The interception algorithm for them is quite similar to what we have seen before:

- a) locate a handler for given file descriptor in our map with **read** lock taken;
- b) if we find nothing, we immediately call the original syscall on a given file descriptor and return from the hook;
- c) otherwise, we pass the vector of buffer to our handler and let it do its job.

2.3.5.5. Read and seek syscalls

Due to the fact that our library support only append-only files any seek breaks this invariant and is not supported formally, however, it is possible that after several appends to empty files application might unexpectedly seek, same applies with

unexpected reads because we do not store read-only files in our map when they are opened, so in order to provide best effort we treat these syscalls the same way we treat positioned writes: flush any present caches, remove association and proceed with original syscall.

2.3.6. Handler architecture

2.3.6.1. General design

Because we deal with append-only files we use the cache to represent only suffix of our file, so, the general model is shown on figure 13 below:

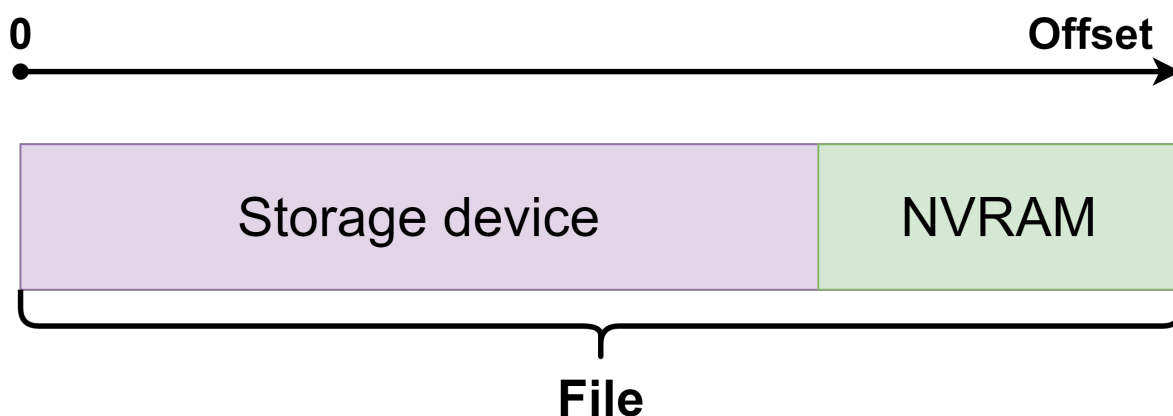


Figure 13 – General scheme of cache

On the surface level, we do two things: 1) append data to our cache, 2) flush data from the cache to disk. However, taking in consideration ability to flush data in the background and durable nature of memory, that we are working with, leads to the list of challenges that are never present in DRAM caches. So, let's take a look at main aspects of this cache.

2.3.6.2. Creation

The first problem that we encounter is a creation of the cache. There are two main challenges here and both of them are somewhat related to the durable nature of our cache:

- we have to allow caching for files that were already created and filled with some user data prior to any use of our library;
- in contrast to DRAM caches, we do not start from empty state every time the handler is created, we need to take into the consideration previous usages of our library.

In order to overcome these challenges we need to solve the identification problem. We need to identify which portion of NVRAM is related to which file being cached. This involves solving two main subtasks:

- In DRAM, this mapping is done with the help of pointers and allocators that are reset every time program is rerun. In our case, this mapping must be usable between different invocations, which for example prohibits use of file descriptors;
- There various ways to open the same file which involve different paths and usage of symlinks.

In order to solve this, we remember about the file identification inside a file system — inodes. As previously discussed, each inode has its unique number that file system internally uses while working with its files and this number and this number is valid as long as file exists. However, this number is unique only within one physical file system, so, we need to have id of device that contains this file system [17]. In Linux, both these numbers can be obtained by a `fstat` system call, that allows to obtain various metadata about a file referred by the given file descriptor. So, we use those numbers to compose filename that used by `libpmem` to create mapping, which solves both our subtasks.

Second problem is to recover from the previous state: which part of NVRAM is a valid suffix of user data and which is not. In order to do so we store special header which contains following info:

- **Flushed position** that shows how much data we flushed to original file on disk;
- **Written position** that shows how much data was written to this file according to outside world;
- **Cache size** that show the amount of cache that was initially configured for this file;
- **Padding bytes** in order extend our header to cache-line size, so its write to NVRAM if always atomic.

When we create a new instance of the cache we firstly write this header, by writing original file size to first two fields. However, the presence of the file does not guarantee correctness of its header, one of the reasons for this is fact, that `libpmem`

does not guarantee contents of newly created mapping, so in order to address this issue we write additional cache-line sized byte string with unique, but always same contents in order to mark correctly created files. This leads to the following metadata structure of metadata:

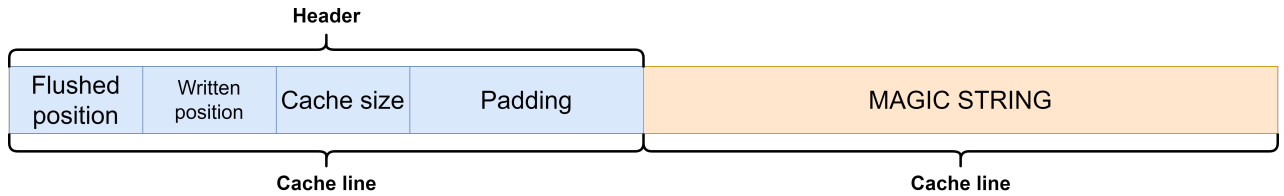


Figure 14 – On-disk layout of metadata

So, after reading the header, we read the magic string in order to ensure correctness of the information in the header. If this check fails we treat this file as fresh new and proceed to initialize it once again, otherwise, we truncate the file to the flushed position in order to remove any junk data. After this integrity of metadata is maintained by cache logic itself.

2.3.6.3. Writing

Generally, the writing in our handler consists of two actions: writing a new user data to the logical end of our cache and writing a data to a file from the logical beginning of our cache. It is also important to note that the second type of writing might be performed asynchronously, which brings an additional complication to this process. In order to do so we use an additional thread that handles writes to original files and correctly updates metadata. However, this requires the synchronisation on the access to the handler. We do it with a simple mutex associated with a cache header.

In order to allow this queue-like access to our cache we implement it as a cyclic-buffer with fixed side. This allows us to allocate NVRAM memory only once and ability to add new data or removing some without any movement of data, that is already in buffer. To be more specific:

- The amount of data in the buffer is calculated as the difference of `header.written_pos` and `header.flushed_pos`. Based on the nature of these positions we always assume that the first one is greater or equal to the second.
- We use offset of data in the file in order to locate data in a buffer to minimise persistent metadata size. This means that the data starts at

`header.flushed_pos modulo header.cache_size` and ends at `header.written_pos modulo header.cache_size` possible turning around the end of out mapping.

Main writing function take an input similar to `writew` syscall and iterates over array of buffers by performing the same writing routine for each buffer. It can be summarised in the following steps:

- a) Get a copy of the header by reading the real one with the lock;
- b) If the buffer does not have enough space for a new input we do following:
 - 1) Explicitly flush all data to a disk;
 - 2) Update flushed position for local copy of header.
- c) Write input to the NVRAM cache;
- d) Update the written position for a local copy of the header;
- e) In case the buffer reaches critical capacity try to initiate a background flush to the original file.

Due to the limitation on single-thread access to every cached file we can be sure that our cache buffer is never overflow because the new data to it can be only added inside this function and its natural precondition is having a buffer not to be overflow too. We add new data only on step **c** and because of checks on previous steps we can be sure that after this buffer is not overflown too, so after this function buffer is never overflown too. The writing does following:

- a) Write a prefix of the input that fits into the space from `local_header.written_pos modulo local_header.cache_size` till the end of buffer.
- b) Optionally, write any leftover starting from beginning of the buffer.
- c) Flush any caches for the written data to the device.
- d) Increase written data position in header and persist it to NVRAM with the lock taken.

It is worth noting that because we write data in order to persist after the write is complete we use `pmem_memcpy_nodrain` allowing us when possible to bypass any process caches, but do not drain hardware caches. There is no need to do so

before all writes are complete because only after writing all input data we update our metadata.

Now, I take a closer look at step **e**. This step allows us to have synchronous writes as fast as NVRAM as long as we are writing not too much data per unit of time.

In order to issue tasks and communicate with our background thread we use a future — construct that acts as a proxy for some concurrent computation [1]. It has several function that are interesting for us:

- **Validity check**: allows to check without blocking if this future is related to some concurrent computation;
- **Getting result**: allow to block on valid future and wait until computation finishes and produces some result so we can fetch it;
- **Checking result**: allow to check without indefinite blocking if valid future already holds ready result or not.

Our handler always holds one instance of the future that relates to the task of flushing data to the original file and uses it to conditionally launch such task if last steps of our writing routine decides to do so. To be precise, function `try_async_flush` takes the current written position (`w_pos`) as input and does the following:

- a) Check if the future is valid and has not obtained result yet. If it is true this means that there is already a flush happening and we do not need to do anything, so, we return;
- b) With the taken lock, we read an actual flushed position from the header to local `f_pos` because concurrent flush might be completed at this point;
- c) We check that there is any data to flush, and if not we return immediately without doing anything;
- d) After passing all the checks, we assign the handler's future to a newly launched task that will write data from `f_pos` to `w_pos`.

It is important to note, that `try_async_flush` function is the only place where the handler's future is accessed. This function itself is only called in functions that are called when the original file is accessed, and by our library limitation it could be done only by one thread until file is closed. Hence, there is no concurrent access

to the future itself which could be devastating because instances of future are not designed to be thread safe themselves.

This flushing task itself is fairly simple:

- a) Write a prefix of the data into the buffer between `f_pos` and `w_pos` that resides before the end of our cyclic buffer;
- b) Write leftover data in the buffer that resides starting at the beginning of cyclic buffer;
- c) Issue `f_sync` for the underlying file;
- d) With the lock taken, increase the flushed data position in header and persist it to NVRAM.

One thing that we have to mention is that the user data in NVRAM is accessed without any explicit synchronisation, because any new user data is written to different segments of circular buffer and writes to our segments are explicitly ordered before this task by launching this task itself from thread that accessed original file.

With all that said, an explicit flushing that might happen on the step **c** of the writing routine can be expressed in the following simple code:

Listing 5 – Possible flushing function in handler

```

1 void wait_flush_task() {
2     if (!flush_task.valid()) {
3         return;
4     }
5     flush_task.wait();
6 }
7
8 void flush() {
9     wait_flush_task();
10    try_async_flush(header.written_pos);
11    wait_flush_task();
12 }
```

One notable detail in listing 5 is that we access the header on line 10 in `flush` without any protection by locks. It is justified because the only thread that could concurrently access header is the thread that performs flush to original files, but we explicitly waited for this thread to finish its task on previous line, so no concurrent access is possible here. Also, there is a possibility that on line 10 we have already flushed all data to the original file and there is no need to do anything. Fortunately,

it is already handled inside `try_async_flush` as mentioned on step **c** of explanation of its logic.

The last topic to address related to writes is the performance concerns one might have due to the lock inside our handler. We need it to correctly read and write header which might be accessed by multiple threads, it is could be seen that every time (with one exception commented above) we access it we protect ourselves by this lock. However there are several factors showing that this shouldn't noticeable affect our performance:

- There are maximum 2 threads that might want access shared header concurrently, so the contention should be minimal.
- Our background thread accesses the header only once during the execution of a flush task. Those tasks are either heavy and are issued only once in a while, so contention on lock is even smaller.
- Every time, a thread accesses the header, under the lock, for a very short time in order to perform fast actions such as reading or writing header and nothing else. So, any real concurrent attempt to obtain the lock should be success within the short time using spinlock fast-path used in most modern locks.

2.3.6.4. Releasing handler

Last thing to mention about our cache is how it could release control of a file when the file is closed, so it could be used later. We perform the following actions:

- a) Explicitly flush all cached changes to original file by calling `flush` which was described below.
- b) Remove any mapping to the file's cache in NVRAM by calling `pmem_unmap` from `libpmem`.
- c) Remove file that represents persistent state of our cache which was created by `pmem_map_file`.

Note that our handler does not close the file descriptor that refers to original file because it could be later used by the original program and should be closed only by this program.

Conclusions on Chapter 2

In the second chapter, I presented a detailed description of our library that uses NVRAM to speed up synchronous writes to append-only files. To be more precise:

- I described the technologies used to work with NVRAM and intercepting program's communication with an operating system.
- I showed how we implemented such interceptions so no changes to source of original code are required to take advantage of our cache.
- I presented a detailed description of a handler that is responsible for handling access to a particular cached file which includes:
 - a layout on disk;
 - details of creating such cache even for already existing files;
 - thorough description of how we handle writes and release control, which includes details on interaction with NVRAM, handling concurrency and communication interception runtime.

CHAPTER 3. ANALYSIS OF LIBRARY

3.1. Performance

In order to set up some expectation for our approach, we compared synchronous writes to NVRAM with writes to SSD. In order to do so we turned to FIO. FIO stands for "Flexible Input/Output," and it is a benchmarking tool that is commonly used to measure and evaluate the performance of storage devices and systems. FIO is designed to provide detailed and accurate information about various aspects of I/O performance, including throughput, latency, and IOPS (Input/Output Operations Per Second).

FIO is highly flexible and configurable, allowing users to define a wide range of workload patterns and parameters to simulate different types of I/O operations. It supports various I/O engines, including synchronous and asynchronous I/O, and provides support for different types of I/O patterns such as sequential and random I/O [11]. In our case we configured it to use classical writes in sequential manner and syncing file after every write to make in synchronous. We present the following result:

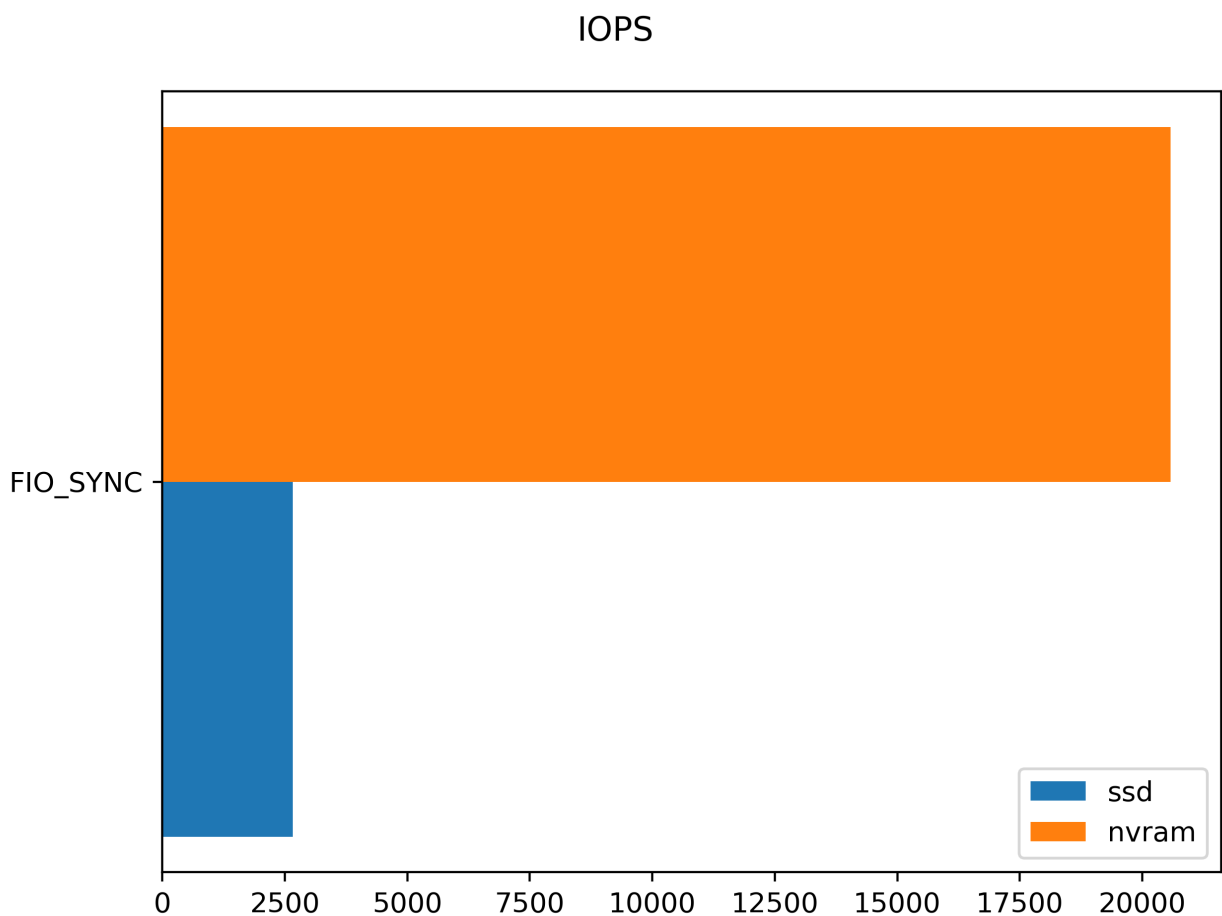


Figure 15 – Benchmarking sequential synchronous with FIO

As expected NVRAM dominates in terms of input/output operations per second (IOPS) in workload that requires sequential access with high durability. However, this experiment only represents a synthetic application that utilises such a workload. We want to test some real-world applications that would benefit from our cache. As mentioned in chapter 1 one popular type of such applications would be a database management system. This system provides the transnational property to database queries, and one core component for doing so is a database journal. It is used to provide the durability property and transactions which include the recovery from unexpected crashes.

For this reason we chose LevelDB. LevelDB is an open-source key-value storage library developed by Google. It provides a fast and efficient way to store and retrieve data based on a unique key. LevelDB is written in C++ and offers a simple and lightweight API, making it popular for embedding in various applications or even building other databases on top of it [19]. Based on their official documentation we derived few key features:

- **Memtable**: LevelDB utilizes an in-memory structure called the Memtable to handle write operations efficiently. The Memtable is an ordered, write-optimized data structure implemented as a skip list or a red-black tree. It holds recently written key-value pairs in memory and allows for fast write operations.
- **Immutable Log (Write-Ahead Log)**: LevelDB maintains an on-disk data structure known as the Immutable Log or Write-Ahead Log (WAL). The Immutable Log serves as a crash-recovery mechanism and keeps track of all write operations. Every modification to the database is first written to the Immutable Log before being applied to the Memtable. In the event of a crash or restart, LevelDB can recover the database state by replaying the write operations from the Immutable Log.
- **SSTables (Sorted String Tables)**: LevelDB stores data on disk in sorted, immutable files called SSTables. An SSTable is a collection of key-value pairs sorted by their keys and compressed for efficient storage. As the Memtable reaches a certain size, it is flushed to disk as a new SSTable. Multiple SSTables can coexist, and they are merged periodically to maintain a compact and efficient database representation.

These features confirmed our initial assumptions of it being one on those applications that should benefit from our cache. LevelDB supports different durability guaranties, however, I am interested only on modes with strongest guarantee — if one received a response from database, his data will be present even the crash or power failure happens right next moment. LevelDB provides several benchmarks to test its performance, so we added them to the initial comparison and the results are shown below:

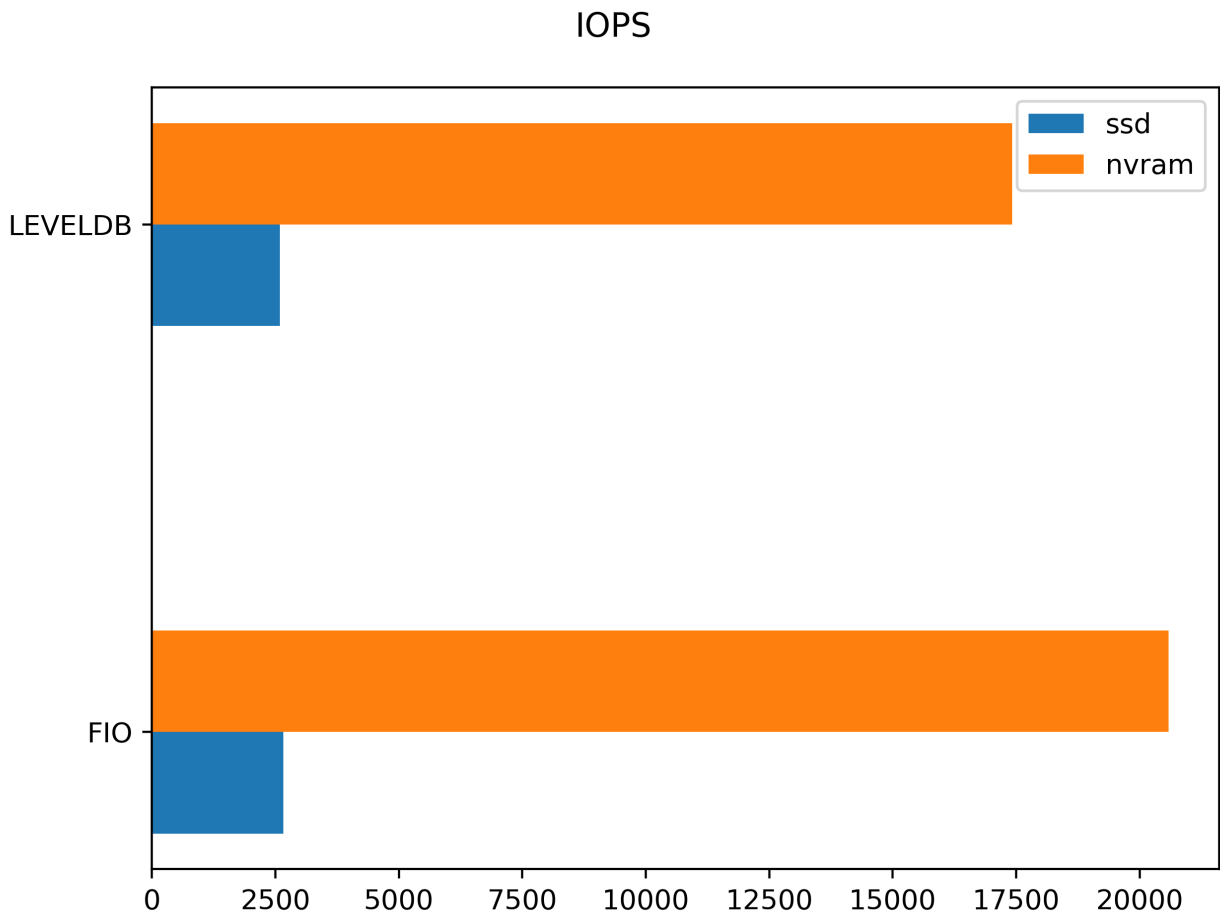


Figure 16 – Synchronous benchmarks of FIO and LevelDB

Once again, as expected, NVRAM shows significantly better performance compared to SSD. However, in this case we cannot be sure that it is attributed to optimisation in workload that is related to database journal. On figure ?? we entirely place everything either on SSD or on NVRAM. It could be possible to that other database structures on NVRAM receive huge benefit compared to SSD for it being faster storage device by itself. However this claim is partially disproved by adding to our benchmarks, that does not require explicitly persisting data to non-volatile

devices after each rather. In this case system returns from syscall as soon as it copies data to its internal buffers in DRAM, which is shown below:

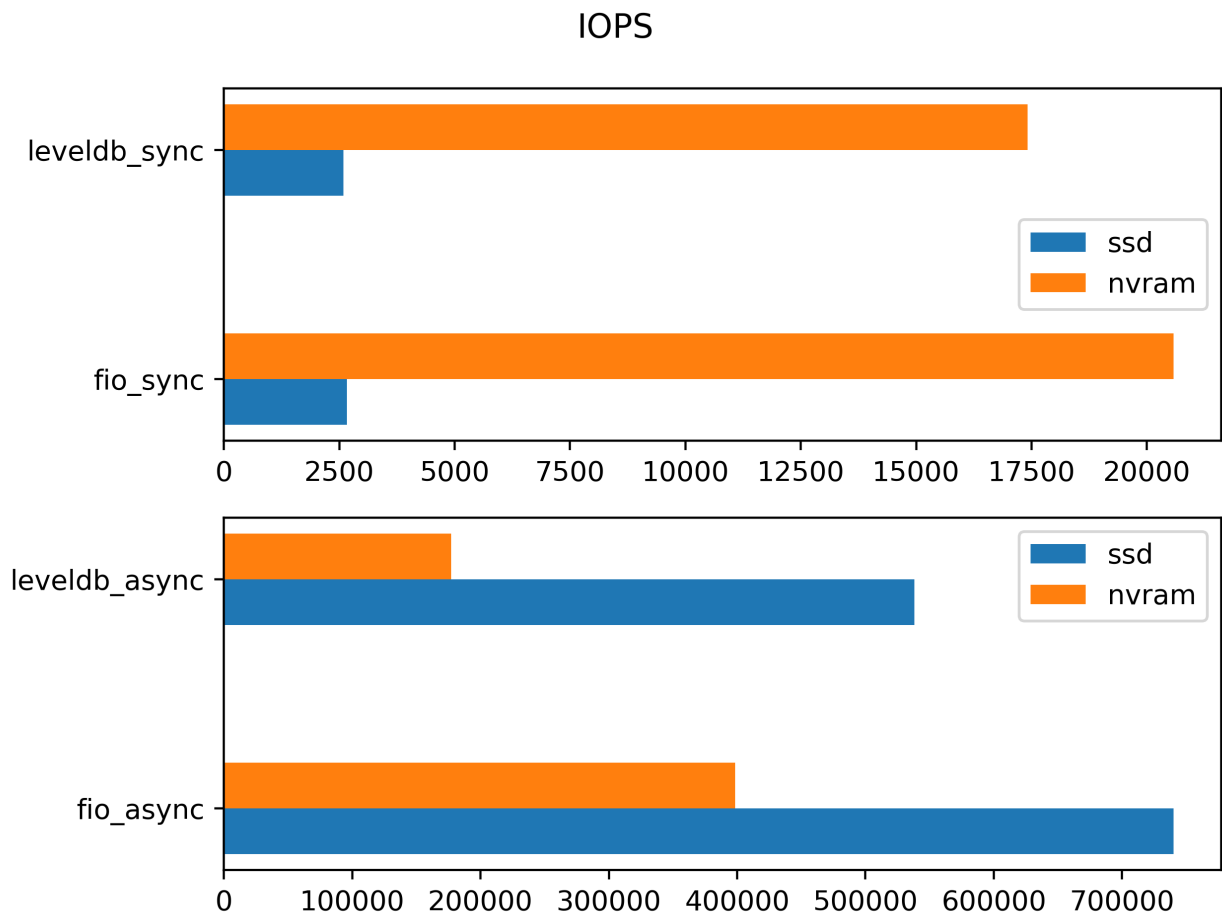


Figure 17 – Synchronous and asynchronous benchmarks of FIO and LevelDB

There are few things to note from the results shown on figure 17:

- Durable writes drastically decrease performance no matter what storage device you use, it must be carefully considered when designing guaranties to your application.
- Despite being somewhere between DRAM and SSD NVRAM is still notable slower when we do not care about the durability that much. Here, a system is allowed to cache without waiting for it to be pushed to a device, which makes DRAM contribution significant due to writes being small in size. However, it is worth noting that it is not slower by orders of magnitudes which is true when we compare DRAM and SSD.
- There is an observable difference in performance between real world application and synthetic benchmarks even if both are significantly bounded by

capability of storage device.

However, to fully ensure that main gains come from boosting our original workload of append-only synchronous writes we want to move only this files to NVRAM. But we encounter one serious problem here: LevelDB places all its files in one directory in a flat manner, meaning there is no hierarchy inside this directory and all files are stored directly in it without any sub-directories, and names of files are generated at runtime. So, there is no easy way to make LevelDB store logs on separate devices.

In order to proceed further, some changes needed to be made to continue our testing. Fortunately, LevelDB uses such thing as Enviroment: the Environment object represents the operating system environment in which the LevelDB database is running. It provides an abstraction for various operating system functionalities such as file I/O, time, and random number generation. What is import in our case: this object is used to list all files in database's directory. Knowing this we do the following to achieve our goal:

- We modified this handler interface to perform listing recursively and implemented this in its implementations.
- We modified a function that creates name for log-files to allow it to accepts optional name of sub-folder where we want to place it.
- We added new parameter to database settings that allow optionally specify a sub-folder for logs and passed it everywhere to function mentioned in the previous item.
- We added new command-line parameter to benchmarking tools, which optionally sets this setting in database it creates to some predefined name.

With this modifications we were finally able to perform our measurement by doing few preparation steps:

- We created empty directories for our databases ahead of time: two on SSD and one on NVRAM.
- We created a special directory for log files on NVRAM.
- In one of our SSD directories we created symlink to directory from previous so when log file in it is created it will be put on NVRAM.

After this we were able to add the third, **hybrid**, type of an installation to our benchmarks and as expected database performance in synchronous mode was mainly restricted by its log files writes to disk with insignificant influence of location of all other files which is shown below:

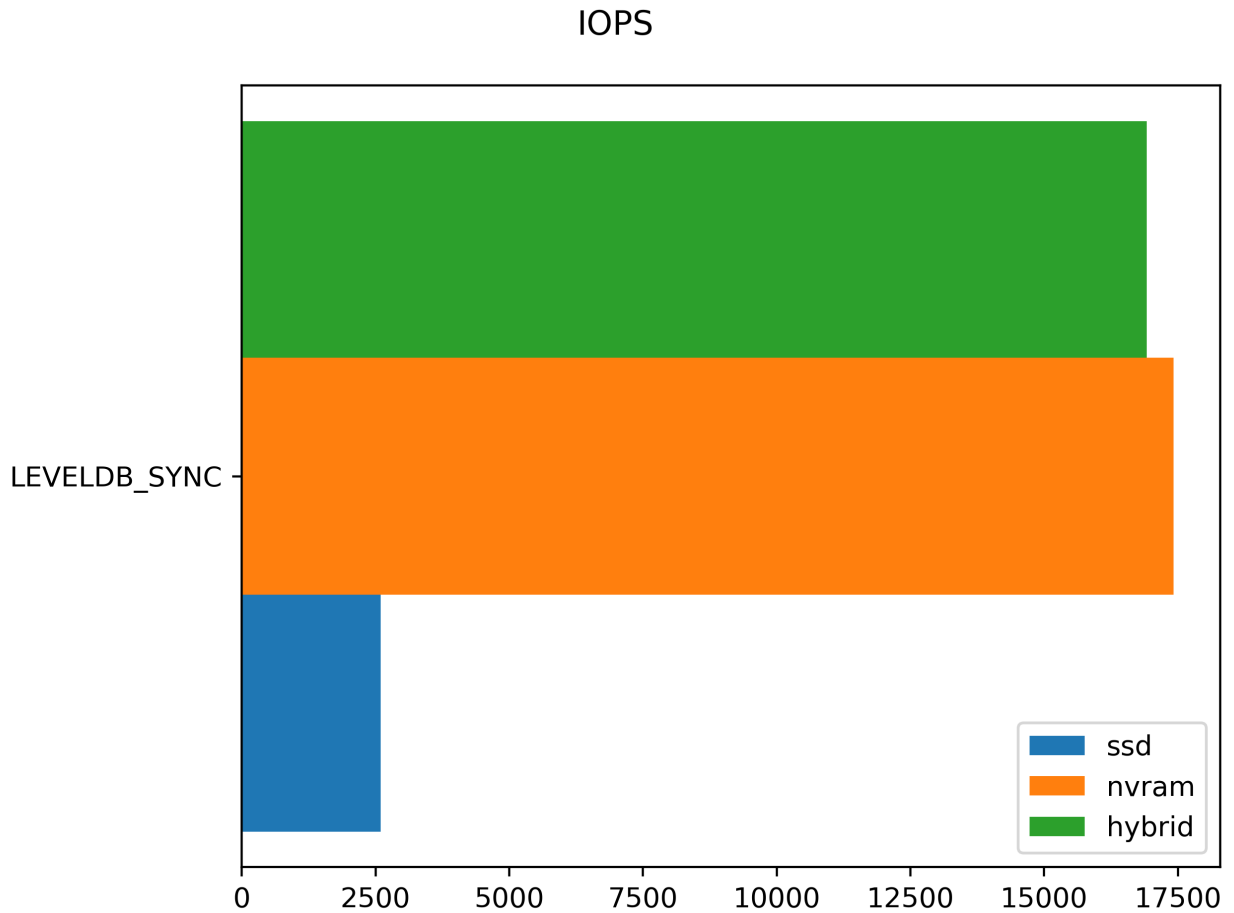


Figure 18 – Synchronous benchmarks of LevelDB on SSD, NVRAM and hybrid mode

With this confirmation of our claim we were finally able to have some reliable baseline for our test.

Before we proceed further it is important to note that the performance of the original benchmark and our library may change depending on their parameters. One very influential parameters is unsurprisingly size of “write cache”, which may have different meaning for our library and for LevelDB itself. However, one of the most important factors is the relative size of cache compared to an original file. Firstly, lets look at situation when our file is relatively small, but still holds enough data for several megabytes of data. It is presented below on figure 19 and shows significant boost in synchronous performance compared even to performance on NVRAM:

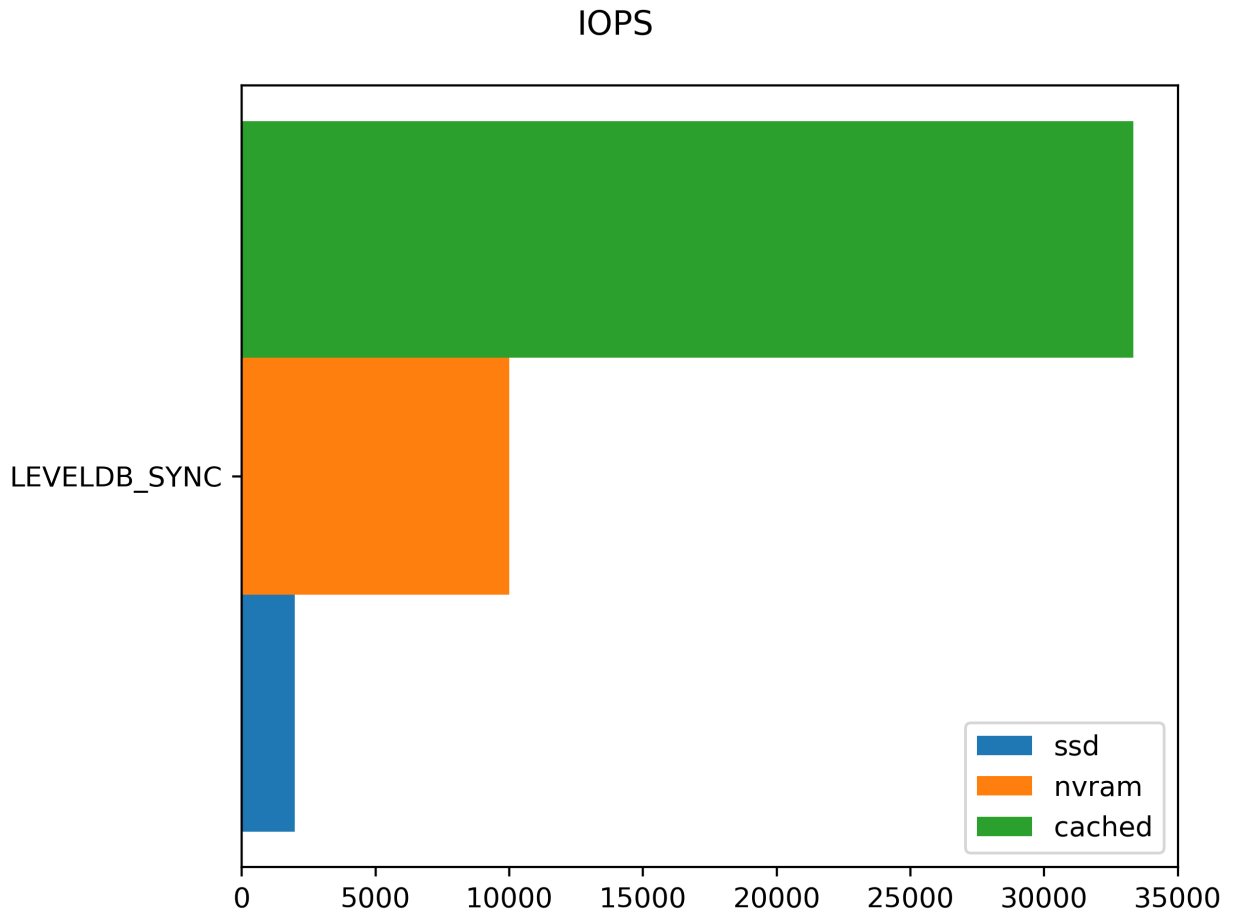


Figure 19 – Benchmarking with small cache compared to original file

This success might be attributed to several factors:

- We perform only one allocation on NVRAM, which we do initially by creating our handler. All writes to NVRAM are performed to already preallocated space, so there is no complicated search for free space which happens when new data is appended to a regular file;
- When we write to NVRAM we attempt to do so directly from user-space without switching to kernel, so there fewer context switches that might entirely reschedule current process;
- Our cached logic is much simpler compared to fully fledged file system, so naturally it is executed faster;
- Cache attempts to push all heavy operations with original file system to background replacing it with relatively small synchronisation inside our handler.

However, when we decrease the size of the database's write buffer making our cache comparable in the size to the database log file our performance boost even further as shown below:

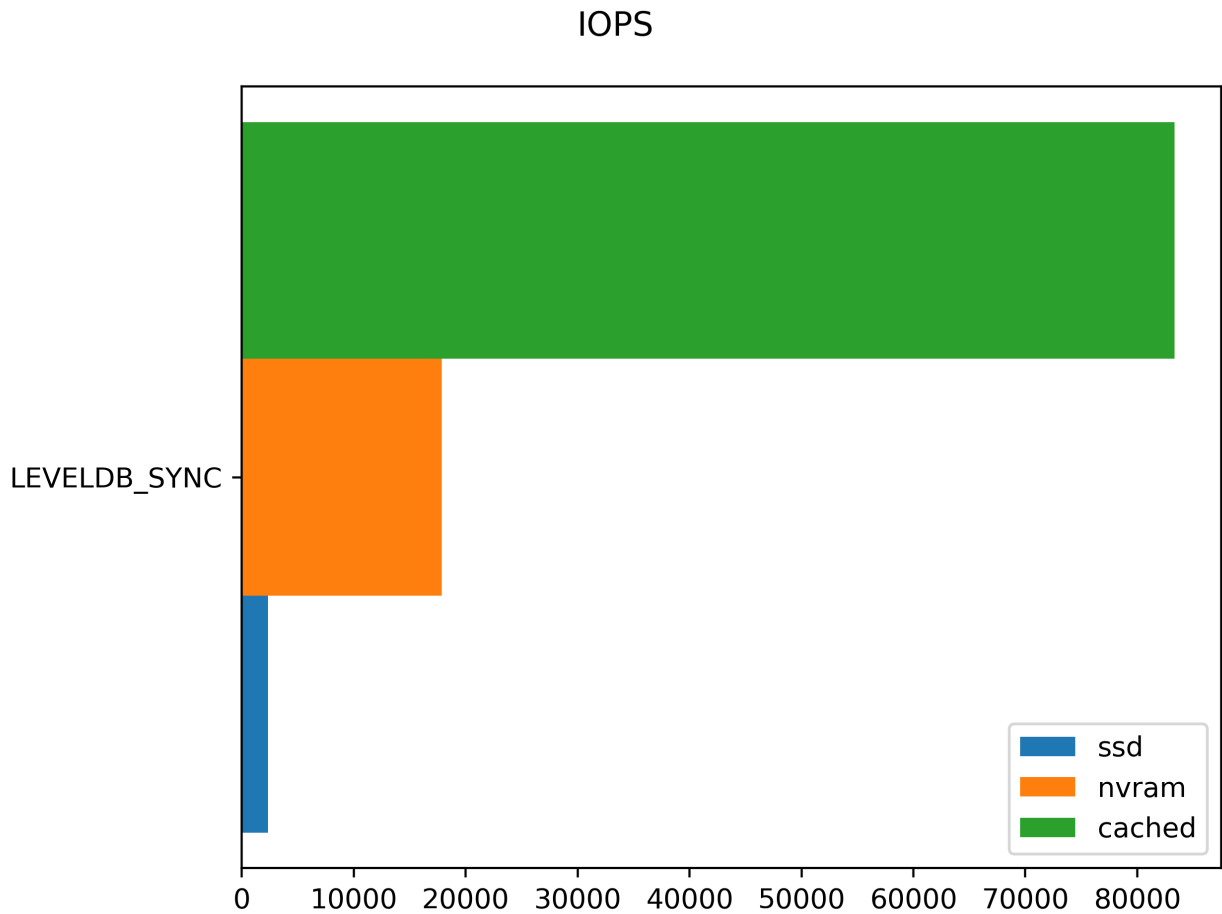


Figure 20 – Benchmarking with big cache compared to original file

One last thing to mention regarding the performance is that some persistent cache are already present before in forms of lvmcache [20] or bcache [6]. However, they work on the block level communicating with block devices. In our case, this required configuring NVRAM as a block device and initial tests show that despite providing boost compared to original SSD benchmark results were significantly slower compared to NVRAM. This is probably the result of the fact that those solution were not written with consideration of NVRAM features and were targeted for regular block devices, which require communication with kernel driver. In addition we already know, that NVRAM performs worse in block device mode due to additional bookkeeping required to provide such interface.

3.2. Correctness

Now, let us take a closer look on the algorithm inside our handler in order to analyse its durability property. We have to keep in mind, that all functions but one are called from one thread so we don't have to worry about their interleaving and this one function access shared state only under the lock.

At first, let's look at the creation of our handler which was discussed in section 2.3.6.2, its part that includes any persistent state could be summarised to following:

- a) Open or create file on NVRAM and map it in memory to some pointer `ptr`;
- b) If we asked to truncate (open empty file) do following:
 - 1) Write file info to the header in memory;
 - 2) Persistently copy the header to NVRAM;
 - 3) Persistently copy the magic string to NVRAM;
 - 4) Return to the caller.
- c) Otherwise, read the magic string by the correct offset from `ptr` and match it with the correct one.
- d) If magic does not match cache file is not considered valid, so once again to the following:
 - 1) Write file info to the header in memory;
 - 2) Persistently copy the header to NVRAM;
 - 3) Persistently copy the magic string to NVRAM;
 - 4) Return to caller.
- e) Otherwise, read the header from `ptr` to memory.
- f) Truncate the original file to the flushed position that we found in the header.
- g) Return to the caller.

Let's list steps where we perform persistent operations and analyse what happens if we crash here:

- **Step a:** If a crash happens after or during this step we either did nothing to our persistent state or created some uninitialised file, that does not have magic string on a needed position.
- **Step b.2:** This step is atomic, so we can crash only after its completion or have no effect from this step. If this happens we either wrote header to some junk file, so it would be discarded and reinitialised on step d, or we wrote new header to the previously existed cache file effectively finishing the required truncation.

- **Step b.3:** This step is also atomic. If the cache was newly created this makes this cache file correct, otherwise, it works as a no-op for an already correct cache file. So even if we restart on this file without truncation it will report the current metadata for the empty file.
- **Step d.2:** This step is also atomic. Here we know that we deal with a junk cache file. If we crash here a cache file will be still considered junk and will be either truncated on step b rewritten again on step d.
- **Step d.3:** This step is also atomic. At this step the file contains correct metadata from previous step, so if we crash here we end up with the correct file that passes check on step c.
- **Step f:** At this step, we already have the correct cache file so any data that we truncate is considered uncommitted and we only do so, to allow future writes to original file to start at the correct position in the file.

Now that we have a correct state on NVRAM and an original storage device we can proceed further and start writing. There are several parts of it which were described in section 2.3.6.3. We start with the flushing routine in the background thread. By the precondition, it writes data that is already persisted to NVRAM that will never be rewritten by the main thread. So, if the crash occurs anywhere before we acquire the lock for the header we might only end up with some uncommitted data in the original file that will be truncated during creation of the header on the restart. After acquiring lock we do two actions:

- a) Update a flushed position for the header in memory. This action is not persisted and is protected by lock, so it does not affect the main thread or our persistent state.
- b) Persistently copy the header to NVRAM. This action is once again atomic, so if we crash right after it before releasing lock it ends up with the correct new metadata in NVRAM because before this action we ensured that this data is persisted on disk.

So after we have a correct flush routine we should take a look at `try_async_flush` that launches it. This function itself does not change the persistent state but dictates how the flush routine will do it. Written position in the header in DRAM can be changed only by main thread. Our current execution is also in the main thread and relying on correctness of our caller we can be sure, that

written position there is correct and is not changed concurrently. We also know from the correctness of the future, which is proved out of scope of this work, that there are no concurrent flush tasks, so we acquire a lock to access the flushed position only to synchronise with the changes that were done to it before by possible previous flush task. This concludes that the flush task that we launch gets correct offsets and we can move on.

The second write that happens by our handler is writing data to NVRAM. By the precondition, it writes data that is logically right after the written position in our handler and this position can not be changed in our background thread so it always stays the same during this function. We also assume by the precondition that logically there is enough place for this data. With this information crash analysis is similar to our push routine. If any crash happens before we persist written data and acquire the lock we will not even notice this because metadata did not change and our cache file's actual size never changes. After this we perform two actions similar to the flush routine:

- a) Update a written position for the header in memory. This action is not persisted and is protected by the lock, so, it does not affect background thread or our persistent state.
- b) Persistently copy header to NVRAM. This action is once again atomic, so if we crash right after it before releasing lock it will end up with correct new metadata in NVRAM because before this action we ensured that data is persisted on NVRAM.

A flush function is fairly simple in terms of crash safety. It is called from main thread and so line 9 in listing 5 is as correct as the flush task is correct, because it only waits for its completion. The next line is also as correct as the function it calls. Here there are no concurrent changes to header if we are able to see its recent state because we are synchronised by future on which we wait on previous line. And finally the last to line 11 applies the same logic as for line 9.

Finally, let's take a look at a writing routine, that ties everything together. Firstly, by copying the header under the lock we ensure that we are synchronised with all changes that were made to it before and also that we see some previously persisted state because we release the lock only after making persistent changes to the header in memory. Because the only possible concurrent change after this

copy may be increasing flushed position we know that the free space in our cyclic buffer may only increase by the time we decide to write data to NVRAM so all our size check stay correct. After this, if there is not enough space we explicitly flush buffer, which we already proved to be crash-safe, so after this action our flushed position and written position remain, which allows us to change the local copy of the flushed position. After this check, we know that there is enough data for our write to NVRAM and we do so once again in crash-safe manner as discussed above, which allows us to update the local copy of the written position which is identical to one on the header in memory. Finally, we apply our heuristic by determining the current cache load factor using local of header, which in worst case is greater than the actual due to the possibly finished flush task at this point. However, this does not affect crash-safety due to the correctness of `try_async_flush` that we call.

The practical correctness of the implementation of this algorithm in our library was mainly checked by running tests for LevelDB, which include crash recovery of the database. However, the single-threaded nature of our handler allow creating a simple helper test routine that sequentially copies one file to another, that logs every synced position and randomly kills [18] itself, so we can verify that we can read correctly reopen this file and the synced content is identical to original.

Conclusions on Chapter 3

In this chapter, I presented detailed analysis of two main aspects of our caching library — performance and correctness. To be more precise:

- I showed its performance in real-world application — LevelDB, presenting reasoning and benchmarking results that showed promising boost in the performance for the synchronous mode of this database.
- I thoroughly analysed algorithm that is used in the file handler in our library by carefully looking at each step and consequences if crash might occur on this. We also gave some insight on practical testing for correctness of our implementation.

CONCLUSION

I developed a NVRAM based cache for append-only files that allows to significantly decrease the latency of such writes which lead to the improvements in the performance of the writing application. We presented this cache in a form of library with the following properties:

- No changes are required in the source code of the original application in order to take the advantages of our library.
- Multi-threaded applications are supported with limitation that only one thread at a time can access each cached file.
- Very small portions of NVRAM are required in order to gain benefit from using this cache.
- Library attempts best effort recovery when application does not follow indented workload to allow it to continue functioning without the crash or corruption.

This library was tested on real-world database where it showed significant performance benefits and proved its durability property. There are several ideas for the future development including the following:

- Porting this library to different systems and/or architectures because the core logic of cache handler does not depend on a particular environment;
- Supporting more sophisticated workloads which mix appending data with random seeks through file;
- Allowing multiple concurrent readers to coexist with one writer.

REFERENCES

- 1 *Baker Jr H. C., Hewitt C.* The incremental garbage collection of processes // ACM SIGART Bulletin. — 1977. — No. 64. — P. 55–59.
- 2 *Codd E. F.* A relational model of data for large shared data banks // M.D. computing : computers in medical practice. — 1970. — Vol. 15 3. — P. 162–6.
- 3 *Herlihy M. P., Wing J. M.* Linearizability: A correctness condition for concurrent objects // ACM Transactions on Programming Languages and Systems (TOPLAS). — 1990. — Vol. 12, no. 3. — P. 463–492.
- 4 Reliability study of phase-change nonvolatile memories / A. Pirovano [et al.] // IEEE Transactions on Device and Materials Reliability. — 2004. — Vol. 4. — P. 422–427.
- 5 The transaction concept: Virtues and limitations / J. Gray [et al.] // VLDB. Vol. 81. — 1981. — P. 144–154.
- 6 *Wikipedia contributors.* Bcache — Wikipedia, The Free Encyclopedia. — 2022. — URL: <https://en.wikipedia.org/w/index.php?title=Bcache&oldid=1118192575> ; [Online; accessed 22-May-2023].
- 7 *Wikipedia contributors.* Readers–writer lock — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Readers%20%93writer_lock&oldid=1155524641 ; [Online; accessed 22-May-2023].
- 8 *Wikipedia contributors.* Static variable — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Static_variable&oldid=1142077360.
- 9 *Wikipedia contributors.* Thread-local storage — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Thread-local_storage&oldid=1153280594.

- 10 *Wikipedia contributors*. X86-64 — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: <https://en.wikipedia.org/w/index.php?title=X86-64&oldid=1156235183>.
- 11 *Axboe J.* fio - Flexible I/O tester [Электронный ресурс]. — 2017. — URL: https://fio.readthedocs.io/en/latest/fio_doc.html.
- 12 fsync(2) [Электронный ресурс]. — 2021. — URL: <https://man7.org/linux/man-pages/man2/fsync.2.html>.
- 13 *Group S. N. P. T. W.* BTT - Block Translation Table [Электронный ресурс]. — 2023. — URL: <https://www.kernel.org/doc/html/latest/driver-api/nvdim/btt.html>.
- 14 *Group S. N. P. T. W.* libpmem - persistent memory support library [Электронный ресурс]. — 2023. — URL: <https://pmem.io/pmdk/manpages/linux/v1.13/libpmem/libpmem.7/>.
- 15 *Group S. N. P. T. W.* libpmemlog – persistent memory resident log file [Электронный ресурс]. — 2023. — URL: <https://pmem.io/pmdk/manpages/linux/v1.3/libpmemlog.3/>.
- 16 <https://www.kernel.org/doc/html/latest/filesystems/vfs.html>. Overview of the Linux Virtual File System [Электронный ресурс]. — 2023. — URL: <https://www.kernel.org/doc/html/latest/filesystems/vfs.html>.
- 17 inode(7) [Электронный ресурс]. — 2021. — URL: <https://man7.org/linux/man-pages/man7/inode.7.html>.
- 18 kill(1) [Электронный ресурс]. — 2022. — URL: <https://man7.org/linux/man-pages/man1/kill.1.html>.
- 19 LevelDB [Электронный ресурс]. — 2021. — URL: <https://github.com/google/leveldb>.
- 20 lvmcache(7) [Электронный ресурс]. — 2022. — URL: <https://man7.org/linux/man-pages/man7/inode.7.html>.
- 21 mmap(2) [Электронный ресурс]. — 2021. — URL: <http://man7.org/linux/man-pages/man2/mmap.2.html>.

- 22 `syscall_intercept` — Userspace syscall intercepting library [Электронный ресурс]. — 2022. — URL: https://github.com/pmem/syscall_intercept.
- 23 *Tallis B.* The Intel Optane SSD 900p 480GB Review: Diving Deeper Into 3D XPoint [Электронный ресурс]. — 2017. — URL: <https://www.anandtech.com/show/12136/the-intel-optane-ssd-900p-480gb-review/7>.
- 24 *Wikipedia contributors.* Berkeley sockets — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Berkeley_sockets&oldid=1150446063.
- 25 *Wikipedia contributors.* Environment variable — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Environment_variable&oldid=1154374071.
- 26 *Wikipedia contributors.* Phase-change memory — Wikipedia, The Free Encyclopedia [Электронный ресурс]. — 2023. — URL: https://en.wikipedia.org/w/index.php?title=Phase-change_memory&oldid=1153076663.

APPENDIX A. COMPARING LIBPMEM TO LINUX MMAP

Listing A.1 – Example of using mmap to write string to persistent memory

```
#include <stdio.h>
#include <sys/mman.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#define LEN 4096
#define PATH "/fs/myfile"

int main() {
    int fd = open(PATH, O_WRONLY | O_CREAT, 0666);
    if (fd < 0) {
        printf("File \"%s\" could not be open\n", PATH);
        return 1;
    }
    if (posix_fallocate(fd, 0, LEN) != 0) {
        printf("Failed to allocate %d bytes for file\n", LEN);
        close(fd);
        return 1;
    }
    char *ptr = mmap(NULL, LEN,
                     PROT_READ | PROT_WRITE, MAP_SHARED,
                     fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Mapping Failed\n");
        close(fd);
        return 1;
    }
    close(fd);
    const char* str = "Hello, persistent memory\n";
    int len = strlen(str);
    strcpy(ptr, str);
    msync(ptr, len, MS_SYNC);
    munmap(ptr, LEN);
    return 0;
}
```

Listing A.2 – Using `pmem_map_file` to write string to NVRAM

```

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <string.h>
#include <libpmem.h>

#define LEN 4096
#define PATH "/pmem-fs/myfile"

int main() {
    size_t mapped_len;
    int is_pmem;

    char *pmemaddr = pmem_map_file(PATH, LEN, PMEM_FILE_CREATE,
                                   0666, &mapped_len, &is_pmem)
    if (pmemaddr == NULL) {
        perror("pmem_map_file");
        return 1;
    }
    if (!is_pmem) {
        printf("Expected pmem device\n");
        pmem_unmap(pmemaddr, mapped_len);
        return 1
    }
    const char* str = "Hello, persistent memory\n";
    int len = strlen(str);
    strcpy(pmemaddr, str);
    pmem_persist(pmemaddr, len);
    pmem_unmap(pmemaddr, mapped_len);
    return 0;
}

```