NATIONAL TECHNICAL UNIVERISTY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
Division of Computer Science

Performance Study and Flexible Data Placement Policies for NUMA Non-Volatile Memory Architectures

DIPLOMA THESIS
Tofalos Filippos

Supervisor: Goumas Georgios
Associate Professor at NTUA

Athens, July 2023

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
Division of Computer Science

Performance Study and Flexible Data Placement Policies for NUMA Non-Volatile Memory Architectures

DIPLOMA THESIS

Tofalos Filippos

Supervisor: Goumas Georgios
Associate Professor at NTUA

Approved by the three-member examination committee on 20th of July 2023.

.....	.....	.....
Goumas Georgios	Nectarios Koziris	Pnevmatikatos Dionisios
Associate Professor at NTUA	Professor at NTUA	Professor at NTUA

Athens, July 2023

.....

Tofalos Filippos
Graduate of School of Electrical Engineer and Computer Engineer, NTUA

Abstract

The commercial availability of non-volatile memory DIMMs (Persistent Memory or Pmem) has provided the landscape of modern computing with devices that combine large capacity, data persistence and performance on the order of DRAM. These characteristics enable utilizing these devices as either memory or storage, and they are intended to be used in NUMA systems. Regardless, the cost of remote access of persistent memory devices arranged in a NUMA topology is much higher compared to non uniform access of DRAM, and said cost presents certain peculiarities, specifically for write accesses.

In this diploma thesis we study the aforementioned arrangement of persistent memory devices in a NUMA setting, in which the devices are used as direct access (DAX) enabled storage. We examine the role of the ext4 filesystem in placing data among NUMA nodes, realizing that its lack of awareness regrading the underlying NUMA topology can noticeably impact the performance of the devices. Given this, we extend the (meta)data allocation algorithms present in ext4 to be NUMA aware and allow for allocating blocks at devices in the same NUMA node as the writing thread. This extension offers up to two times performance gain in I/O intensive workloads.

Finally, we extend the per-process I/O statistics collection for accesses to the storage layer so as to additionally support collecting this information per NUMA node and for DAX mode as well. Based on these statistics, we outline the development of a userspace library that dynamically places application threads to the suitable NUMA nodes for the purpose of better utilizing the aggregate bandwidth of the persistent memory devices.

Keywords

NUMA, persistent memory, filesystems, Linux kenrel

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Goal

The typical structure of a modern computer includes, among other things, a processing unit (CPU), non-volatile storage media to hold data between power cycles, and (volatile) RAM connected via the bus to the processor.

There are two main problems with this structure: The first problem is that there is a gap between the performance of processors and memories, which is only exacerbated by the increase in the number and speed of processing cores. The second is that there is also a significant performance gap between non-volatile storage media and memory.

The first problem is addressed at the architectural level of computing systems by the introduction of the concept of NUMA. Previously, all processing cores accessed the same pool of memory, quickly draining its bandwidth. In the NUMA architecture, the cores are now divided into groups, the so-called NUMA nodes, each with its own (local) set of memory and storage modules. To ensure system data consistency, an interconnection network is placed between the NUMA nodes. This leads to accesses with uneven performance, depending on whether we access the local memory pool of the corresponding processing core, or if we need data from another node, so mediation from the interconnection network is required. We say we have local and remote accesses respectively.

The second problem can be solved at the level of storage medium technology, and is under constant investigation by the research community. A few years ago, the first non-volatile memory modules became commercially available, namely the Intel Optane DIMMs, a product that promises the feature of non-volatility with performance closer to that of DRAM technology than to pre-existing storage media technologies.

Both the NUMA architecture and the non-volatile memory technology are

aimed at high-performance computing systems, which both need the more efficient use of the memory system provided by the former, but also the faster access to stored data provided by the latter. There is therefore a need to study the way in which the two aforementioned factors interact, in order to appropriately adapt the existing software infrastructure in a way that will prevent any negative synergism between the two.

Specifically, the overall goal is for the end user of a NUMA system to be able to utilize all of the system's resources with as much transparency of the underlying architecture as possible. Therefore, to use the non-volatile memory pool efficiently given non-uniform accesses to it, we need to identify the associated vagaries and pathologies that can arise. The ultimate goal is to incorporate some special case handling or avoidance logic into the implementation of the appropriate abstraction mechanisms provided by the operating system for accessing non-volatile memory modules. In this case we are interested in the function of non-volatile memories as a storage medium, and for this reason the mechanism we aim to remodel is the file system.

1.2 Contribution

In this diploma thesis we properly extended the ext4 file system in order to examine and demonstrate the efficiency of space allocation when it is aware of the NUMA topology of non-volatile memory.

We studied the behavior of non-volatile Intel Optane DIMM devices by identifying a fundamental pathology when they receive sequences of heterogeneous remote accesses. Given this, we appropriately re-engineered ext4's (meta)data allocation routines to result in more efficient accesses when the filesystem is build on a properly configured device that linearly joins all non-volatile devices in the system. This device is obtained through the device mapper driver of the Linux kernel.

A complete methodology was developed to evaluate the correctness and efficiency of the aforementioned extensions, showing that the modified version of ext4 is capable of providing up to two times better utilization of the cumulative bandwidth of non-volatile devices in some I/O execution scenarios, compared to vanilla ext4 and for different ways of connecting the devices.

We chose the ext4 filesystem as it is widely used and offers the option of using the page cache or not at the level of individual files (known as the per-file DAX feature). By providing ext4 with NUMA awareness, we build the needed infrastructure for future study of how to conditionally leverage the page cache to limit harmful direct remote accesses to non-volatile memories.

In order to better facilitate future research on the topic considered in this

diploma thesis, the I/O statistics collection system of the Linux kernel was extended so that, in cooperation with the device mapper driver, it can support the collection of statistics at the level of NUMA nodes, and in a way that is compatible with the DAX feature.

Based on the extension of the IO statistics collection system, we form the basis for a userspace mechanism that collects execution data for selected threads and dynamically places them to the appropriate NUMA node. Although the source code for this mechanism is currently simplistic, this document analyzes how to extend it and what are the related challenges needed to be solved in order to allow its evolution to the necessary extent.

Finally, apart from the already mentioned possible extensions of this work (namely the completion of the userspace mechanism and the conditional utilization of the page cache to smooth out the pathogenesis in non-uniform accesses), we also briefly describe the way in which we could provide NUMA awareness to the accesses of already allocated data blocks.

1.3 Outline of the Report

Chapter 2: Theoretical Background

In this chapter we see in more detail the concept of NUMA architecture, the various memory technologies and how they compare to non-volatile memories, architectural elements of the non-volatile memory modules, and details on the implementation of the ext4 filesystem, the understanding of which contributes to the modification of the file system as described in chapter 6.

Chapter 3: Methodology

In this chapter we present all the available tools and methods used for developing the Linux kernel and conducting our experiments. In particular, we briefly describe the operation of the FIO and Filebench tools, the automated system for running experiments as provided in this work's source code, as well as the configuration for the kernel development infrastructure and the methods of analyzing the kernel's behavior.

Chapter 4: Experimentation

In this chapter, we analyze the experiments highlighting the problematic patterns that may arise in non-uniform accesses to non-volatile memory modules, as a foundation for the motivation for this work.

Chapter 5: Motivation

In this chapter, we present at an abstract level the goals we set for this work based on the results of chapter 4.

Chapter 6: Implementation

In this chapter, we describe the details regarding the design and implementation of the software infrastructure developed, and we list some possible extensions of the userspace component.

Chapter 7: Evaluation

In this chapter we describe the entire process of evaluating the changes made to the ext4 filesystem, with regard to both the implementation's correctness (stability of execution and respect of the desired properties, mainly NUMA awareness) and performance, using the tools of chapter 3.

Chapter 8: Related Work

This chapter presents relevant research on non-volatile memories. In particular, we mention filesystems developed to be tailored to the performance of non-volatile DIMMs, and we also cite work specifically aimed at the performance of non-volatile memories in relation to the NUMA architecture.

Chapter 9: Epilogue

In the last chapter, we summarize the work done in this thesis and refer to its possible extensions.

1.4 Source Code

The source code of the work presented in this document can be found at the following repository:

`https://github.com/PhTof/Diploma`

Oftentimes in this document there will be references to files of this repository. In some cases, references may also be made to the source code of version 5.13 of the Linux kernel instead of the modified version found in our work's source code. It will be explicitly noted which of the two cases we refer to.

Chapter 2

Theoretical Background

In this chapter we will present the basic theoretical background needed for a better understanding of the present work and the computational infrastructure it utilizes.

2.1 NUMA Architecture

Very early in the history of computing systems arose the problem of the CPU-memory performance gap: the ability of memory modules to handle requests lacked significantly compared to the rate at which CPUs can issue memory access operations. This gap only widened throughout the years. The immediate solution to maintaining scalability of computing systems has been the mediation of small, fast caches between CPU and DRAM [21].

However, even the evolution of these intermediate, faster memories could not fully satisfy the needs arising from the development of multiprocessing systems, which led to an ever-increasing number of processing cores competing for memory bandwidth, leading to further memory performance degeneration.

One of the solutions to this problem has been the development of systems where specific subsets of processing cores, corresponding to a NUMA node, have their own, local memory that they can directly access. Processors of different NUMA nodes communicate with each other through an interconnection network, via which data is transferred when a processor needs to access the memory of another NUMA node.

Naturally, the speed at which a memory access request is serviced depends on whether the NUMA node to which the memory unit is attached is local to the requesting processor or not. Therefore, the architecture is called non-uniform memory access (Non-Uniform Memory Access).

Figure 2.1 shows the schematic representation of a 2-node NUMA system,

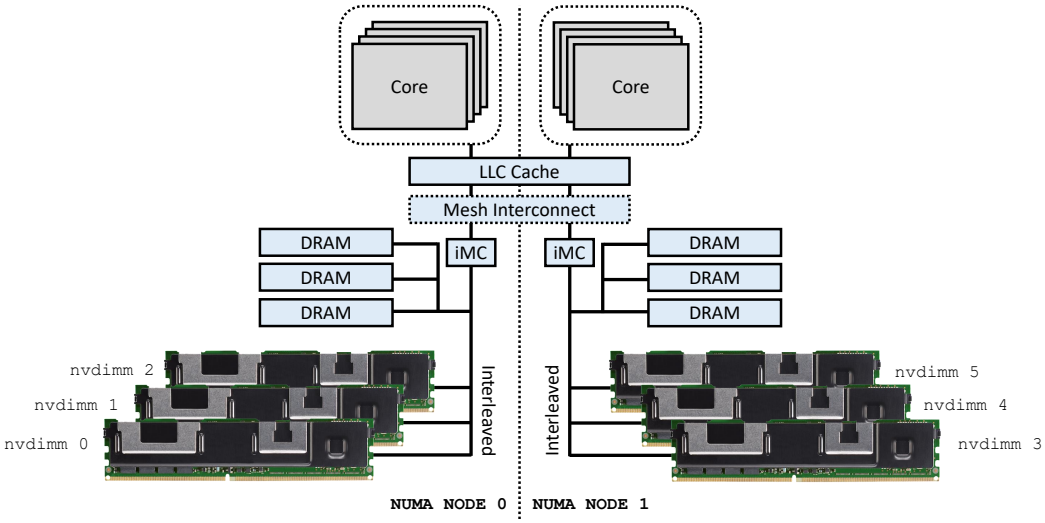

Figure 2.1: Non-volatile memories in NUMA architecture (AppDirect configuration)

presenting how the different nodes are separated at the system architecture level.

2.2 Storage Media Technologies

For ease and completeness of reference afterwards, we list and explain some of the metrics we will use to characterize storage or memory media:

- **Throughput** (otherwise referred to as utilized bandwidth): The rate at which we read or write data to the respective medium.
- **Latency**: The time that elapses from issuing an access request (read/write to the medium) up until it has been fully processed, i.e. the requesting entity is notified upon write completion, or the requested read data has been transferred.
- **Data Volatility**: By the term volatility, we refer to the property of not being able to retain the stored data in conditions of a power outage. The concept of non-volatility follows logically from the previous definition. Volatile devices may be DRAM units, while non-volatile devices are commonly hard disks or NAND flash memory.
- **Density**: The information that can be stored on a unit of physical space of the storage device, measured in bits. This unit can be the total area of the integrated circuits of the device.

In addition, we also have a functional feature, related to the way data is accessed on the device: the so-called data access granularity. This corresponds to the minimum amount of information that can be transferred through A singular access. We are concerned with the terms of byte granularity and block granularity. The first one means that accesses can happen at the level of bytes, or, equivalently, for any range of bytes. By the second term, we mean that accessed data comes strictly as part of indivisible data transfer units, specifically the so called "blocks" of a certain size.

We will see that the concept of granularity is embedded within the logic of some filesystems, as the Direct Access (DAX) feature for devices that support byte addressability.

Before the concept of non-volatile memories, there was a clear separation between commercially available data storage devices and memories. On one hand, we have Dynamic Random-Access Memories (DRAM) operating at byte granularity and offering high bandwidth, low latency, but low density. On the other hand, we have storage devices such as hard disks or SSDs of various kinds. Unlike DRAM, they enforce block granularity and have orders of magnitude lower throughput as well as higher latency, but have increased capacity and the property of non-volatility.

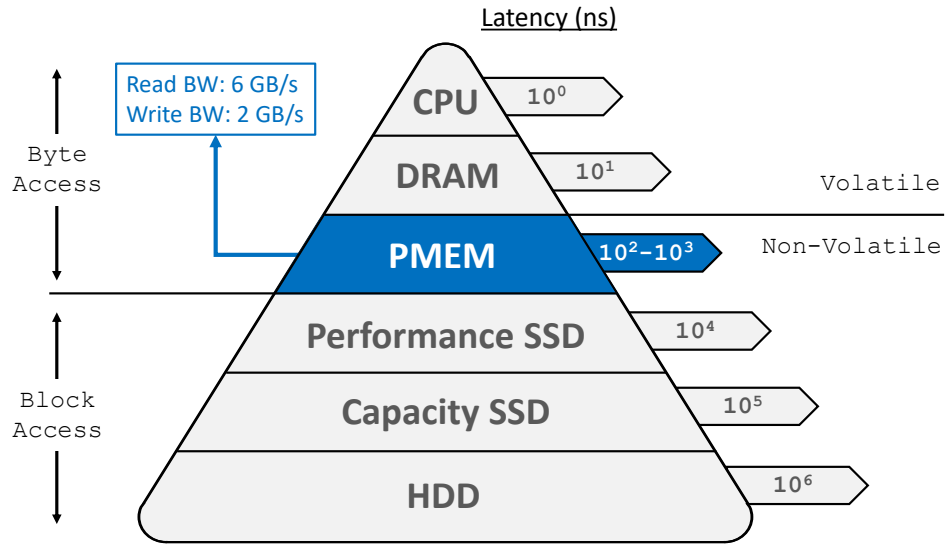

Figure 2.2: Placement of non-volatile memory in the hierarchy of storage media technologies

Between memory (as we will informally refer to DRAM) and traditional storage media there is a significant performance gap if we take bandwidth and latency as metrics. Non-volatile memories tried to bridge this gap by combining

the desired characteristics of each individual type of device. In particular, they have byte granularity, the property of non-volatility, and performance characteristics an order of magnitude close to DRAM. This fact provides non-volatile memory modules with applications both as part of the temporary data storage hierarchy (with DRAM acting as a direct-mapped cache of non-volatile memories, in a mode of operation referred to as memory mode) and as data storage medium (which we will refer to as AppDirect mode).

2.3 Characteristics of non-volatile memories

In the previous section we saw more abstractly how non-volatile memories rank in the hierarchy of various data storage technologies (volatile and non-volatile). Here, we will attempt to present in more detail some elements of their architecture, among others. Although not everything that will be mentioned immediately after is directly used in implementing our work, it provides useful intuition for understanding how non-volatile memories work.

It is important to note that much of the information in this section is based on publication [31].

2.3.1 Performance Characteristics

To further develop upon what has been previously mentioned, compared to most traditional storage media, non-volatile memories achieve higher bandwidth and lower latency by at least an order of magnitude. Non-volatile memories lag behind DRAM in both of the aforementioned metrics [31], but offer non-volatility and higher density. As we will see in chapter *Experimentation*, an important peculiarity of non-volatile memories is the disparity between read and write performance characteristics. In particular, read performance is three times that of write, while these are uniform in DRAM technology.

2.3.2 Communication Interface

Traditional storage media are connected to the computer system either through the SATA interface, or on the physical PCI Express interface through the NVMe logical interface, for the most efficient among them. Intel's non-volatile memories are connected on the memory bus, just like a DRAM unit. The non-volatile memory module's communication interface with the processor's iMC (integrated Memory Controller) is called DDR-T, and it has the same physical characteristics as DDR4 but a different communication protocol [5].

2.3.3 Processor-Medium Pathway Architecture

Our following description is based on the next figure of publication [31].

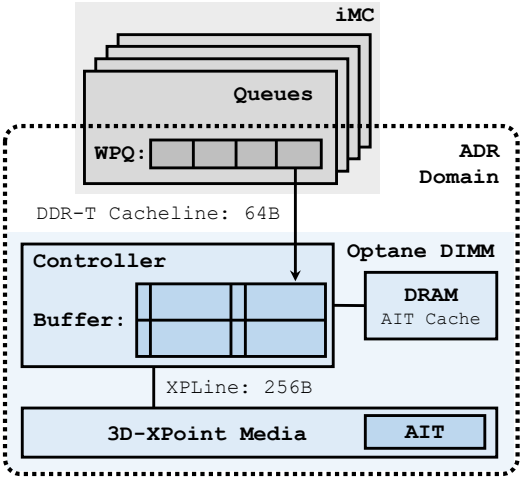

Figure 2.3: Abstract overview of the architecture of non-volatile memories

As shown in figure 2.1, each NUMA node has its own iMC (integrated Memory Controller). Each request to the memories is forwarded from the processor to them via the iMC. This includes multiple read and write queues (RPQs and WPQs). A critical difference of the two is that the write queues sit in the so-called Asynchronous DRAM Refresh domain (ADR). Anything in ADR is guaranteed to survive a system power outage, meaning it will be written back to non-volatile memories in time. The iMC communicates the request to the corresponding non-volatile memory module (Optane DIMM) via the DDR-T interface with a granularity equal to that of a cacheline (64 bytes).

Regarding the operation of Optane DIMMs, we note two important things regarding the 3D-XPoint technology: the first one is that the medium access granularity is 256 bytes (the so-called XPLine in the diagram), i.e. four times larger than that of the DDR-T interface. The second one is that, as is common with other storage media such as SSDs, non-volatile memories also need a wear leveling mechanism to compensate for the limited endurance of the storage medium.

Given the difference in granularity between DDR-T and 3D-XPoint media, among other things, a buffering unit is placed in the device controller, combining accesses to adjacent addresses before writing to the medium (write-combining buffer or XPBuffer in the figure). This aids in more efficiently serving consecutive smaller write accesses, as well as combating write amplification. On the other hand, due to the need for wear leveling, an address indirection table (AIT) is established and a buffer (AIT Cache) is included to smooth its operation.

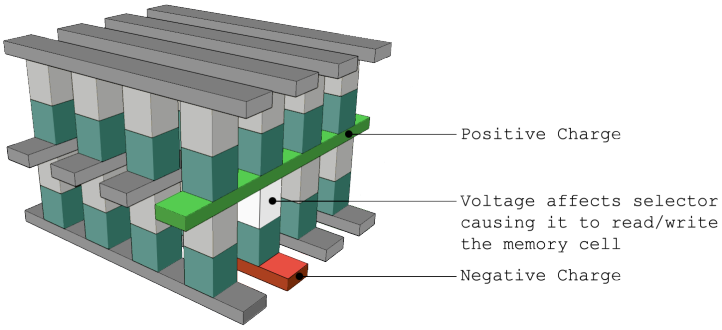

Figure 2.4: The 3D cross-point technology
Image based on figure from the BBC article titled "3D Xpoint memory: Faster-than-flash storage unveiled"

2.3.4 Storage Medium Technology

The non-volatile memory modules utilized for our purposes, commonly known as Optane DIMMs, are based on the 3D Xpoint technology that materialized from the collaboration between Intel and Micron Technology. It was differentiated from previous versions of PCM (Phase Change Memory) technology by offering faster and more stable memory cells, through the use of threshold switches, based on chalcogenide materials [4], instead of transistors as selectors.

The selection of bits to be accessed is done with the aid of the Cross Point structure: for a square array of bits we have on one side (top or bottom) of the array wires-columns, and on the other side we have wires-rows. From the actuation of a specific column and row, a single cell is selected for accessing, as shown in figure 2.4. This structure offers stackability (and is thus characterized as "stackable"), which provides the devices with the property of higher density [3].

2.3.5 Configuring Non-Volatile Memories

We have two basic options for configuring non-volatile memories: the first option, which is the only sensible one for using memory as a storage medium, is AppDirect mode. In this mode, the devices are used as storage, and they can function completely separately from DRAM modules (despite the fact that a filesystem may choose to utilize DRAM as a page cache, but this is software related).

The other option is memory mode, where each non-volatile memory module is paired with a DRAM memory module on the same memory channel. Non-volatile memory appears to the operating system as main memory, and DRAM

takes on the role of a direct-mapped cache of the former. The operation of DRAM as a cache is handled by the processor's memory controller [31].

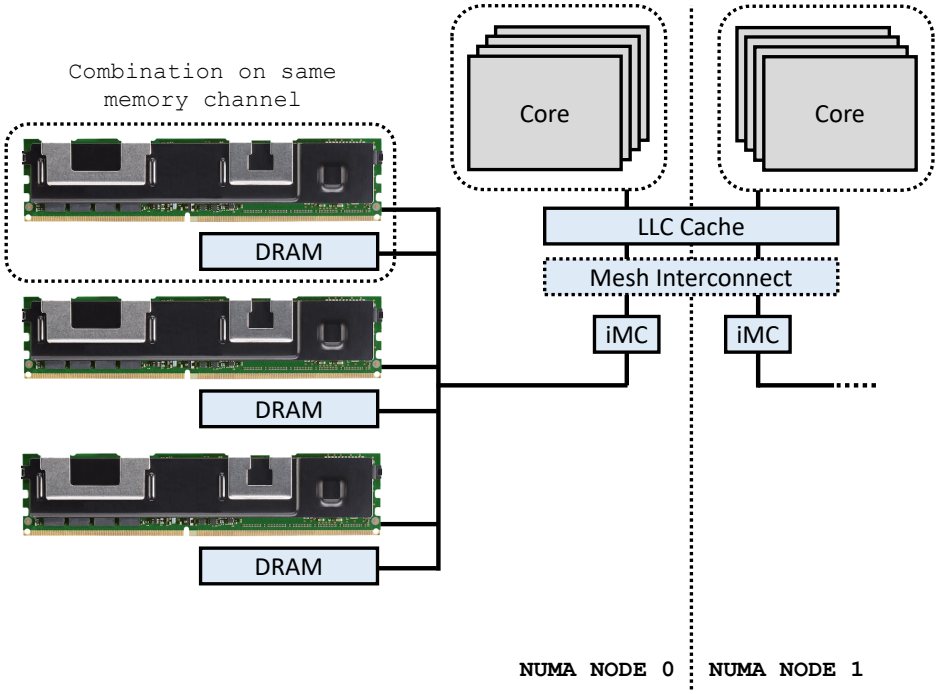

Figure 2.5: Memory mode

AppDirect mode is shown in figure 2.1, while Memory mode is shown in figure 2.5.

2.3.6 DAX Feature of Filesystems

For storage devices, like non-volatile memories, offering a fast, byte-addressable interface, filesystems of the Linux kernel, such as ext2, ext4, and xfs, implement the DAX (Direct Access) feature. When this feature is enabled, the compatible storage medium is accessed directly, without the mediation of the page cache. Correspondingly, with the DAX feature disabled, data accesses go through the page cache, i.e. pages are temporarily stored in DRAM for faster possible future accesses. The page cache needs to write back to the medium any modified pages at some point, in order to keep it up to date.

The page cache offers better throughput for accessing its cached pages since it sits in DRAM. However, its capacity can be significantly smaller than that of the storage medium, and for each page that is not already in the page cache, two copies are required (double copying effect): one to transfer the page to the page

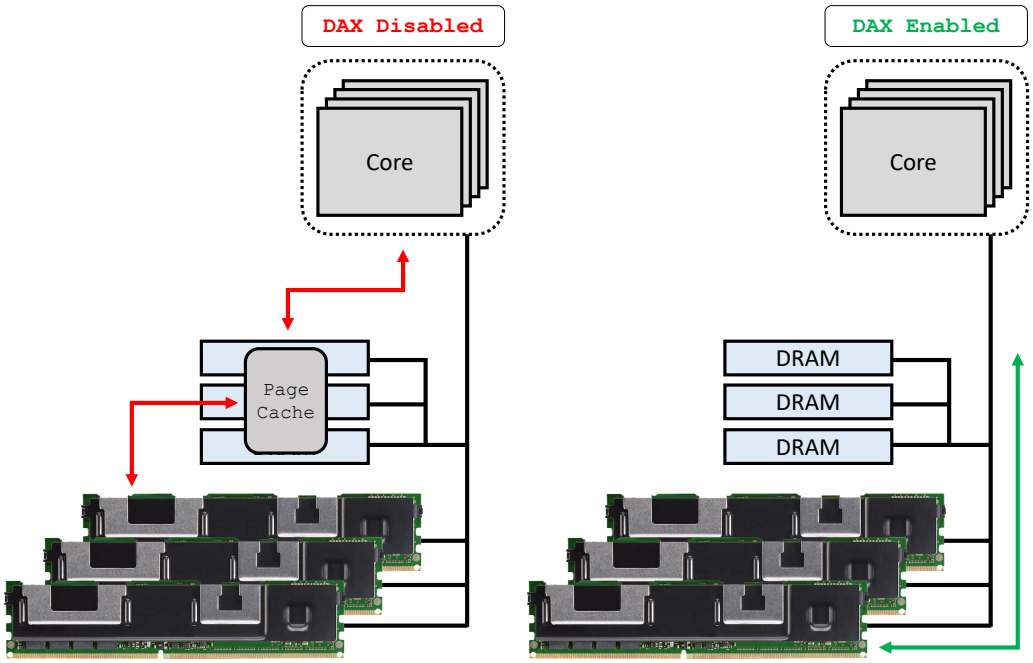

Figure 2.6: How DAX works

cache, and another to transfer the data to the requesting process. Media with comparable performance to DRAM, such as non-volatile memories, may in some cases suffer more from double copying than benefit from the greater bandwidth offered by DRAM.

2.4 Existing Ext4 Optimizations

In this section we will look at some of the optimizations that ext4 employs, which proved to be directly relevant to the code development process of our work. Although typically the purpose of this chapter is to present the theoretical basis our work builds upon, without yet listing details of the work itself, it was considered appropriate for ease of reference to briefly mention for each optimization how it was adapted (if at all) in our implementation. A more detailed presentation will take place in section *Modifications to Ext4* of chapter *Implementation*.

2.4.1 Extent Tree

An important design decision in filesystems is how a file's metadata indexes its data blocks.

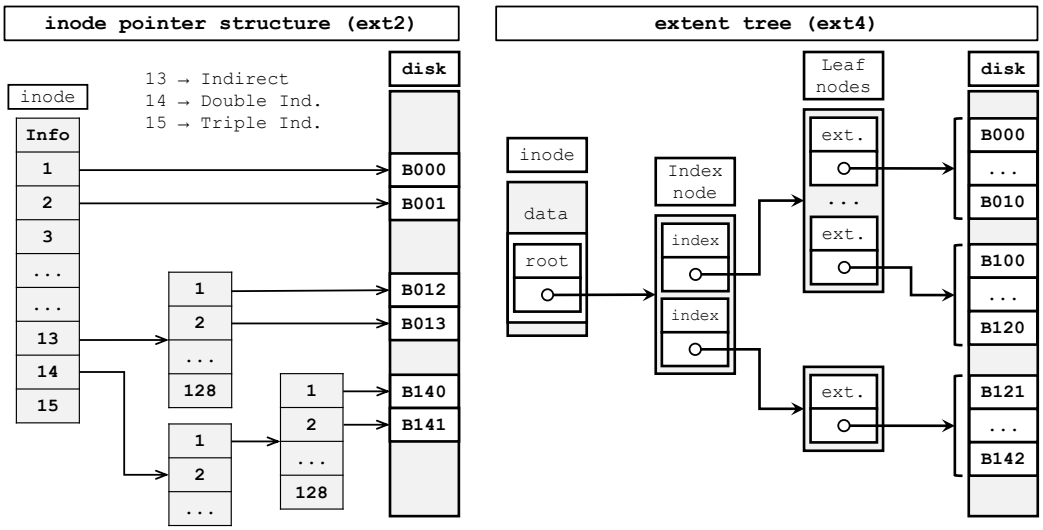

Figure 2.7: File Data Indexing Methods

Historically, in filesystems like ext2, the way data blocks have been indexed is via arrays of fixed size, with each entry of the array being a pointer to some block or another array of the same type, as shown in the left part of figure 2.7. For ext2, files start with an array of 15 entries. Entry 13 points to an array (of size 128 for older systems) that directly indexes blocks. Because of this structure, entry 13 is said to be doing single indirection.

Entry 14 points to an array which indexes arrays like the one of entry 13. So what looks like a 2-level tree structure is formed. Entry 15, which is also the last of the inode's inlined array, indicates a similar structure, but with three levels. That's why positions 14 and 15 are respectively said to do double and triple indirection.

All arrays except the inlined one have fixed size, that of one block, and are only allocated when necessary. That is, the array at entry 13 is allocated when the remaining 12 entries of the inlined array are occupied, the rightmost table of figure 2.7 is allocated when all entries of the single indirection table are occupied, while already having allocated the array of entry 14 in order to do double indirection, etc.

This indexing scheme does not take advantage of the very common case where several blocks are allocated sequentially on the storage medium, to avoid wasting space due to metadata. In general, by indexing block ranges instead of each block individually, we can have a more economical representation of the data blocks of each file. This is precisely the idea ext4 employs, which introduces the concept of an extent tree, a tree structure with leaves that indicate a range of contiguous blocks (the so-called extent). This structure is shown on the

right part of figure 2.7.

While in this work we won't change the functionality of the extent tree in any way, it helps to know the indexing scheme since it's only logical that its structure directly affects how new blocks are allocated. So, we'll see what limitations are imposed on NUMA-locality of allocations due to the extent tree's interaction with other optimizations, and in chapter 10 we will briefly mention how its operation should be adapted to support NUMA-local file overwrites.

2.4.2 Flexible Block Groups

A block group is a fixed sized collection of consecutive blocks. We define groups consisting of block groups, which are called flexible block groups. In each flexible block group, the first block group collects its own and the rest of the groups' metadata. This way, we achieve better metadata locality, and get larger chunks of contiguous blocks of data since metadata is not interpolated as often.

For convenience, this feature is not currently compatible with our additions for supporting NUMA awareness.

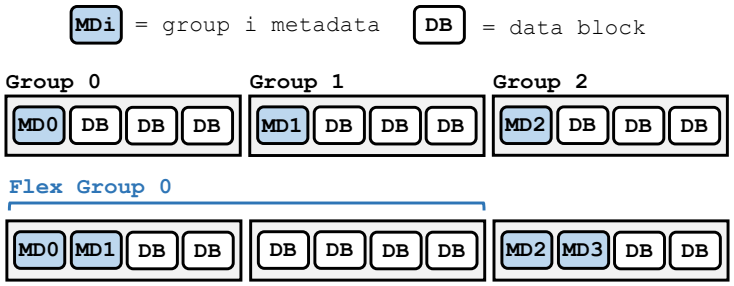

Figure 2.8: Flexible Block Groups: With `flex_bg` enabled, group 0 collects its own metadata alongside that of group 1, thus creating flex group 0.

2.4.3 Locality Group Preallocations

We try to satisfy relatively small requests from a preallocation pool of the requesting processor. This way, we achieve locality and speed in allocation of small files.

We adjust this feature to be aware of NUMA locality, by selecting from the corresponding locality group preallocation pool only preallocations relevant to the local NUMA node.

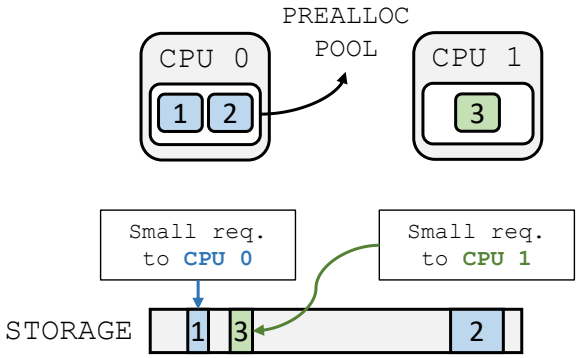

Figure 2.9: Locality Group Preallocations: We denote references to locality group preallocations as numbered rectangles with a different color for each processor. Here, it is seen that each processor consults its own locality group preallocation space for the small request it serves.

2.4.4 Per-inode Preallocations

On each allocation request for a certain file, we bind more space than requested, keeping the excess as preallocation for future use. This way, we obtain more ideally sized regions that often correspond to selected powers of 2, and future allocation requests may be served faster.

In order to make this optimization NUMA aware, for each inode we now keep a separate list per NUMA node, instead of a singular preallocation list.

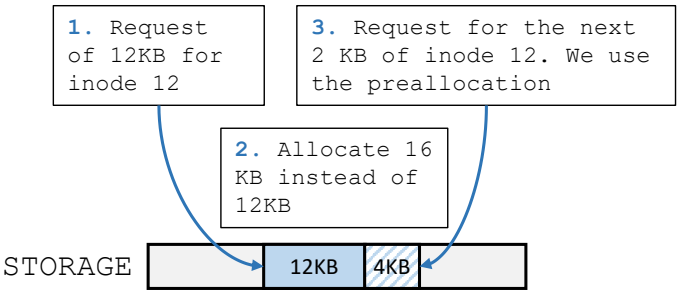

Figure 2.10: Per-inode Preallocations: We have illustrated how a new per-inode preallocation stemming from a request may be used in a subsequent request. The outlined part should be though of as indexed by some list that is part of the inode structure.

2.4.5 Stream Allocation

Relatively large requests do not use the locality group preallocation pool, and we try to start the allocation from the point where the previous stream allocation

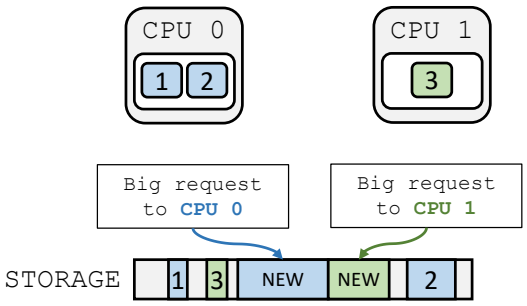

Figure 2.11: Stream Allocation: Chronologically, We have supposed that the request of CPU 0 precedes chronologically that of CPU 1. We see that the allocations of these requests are made in a spatially continuous manner.

ended. In such doing, we get smoother IO traffic, and we avoid the creation of large fragments (storage space between spatially consecutive allocations).

Adapting this optimization for NUMA awareness purposes is straightforward, as it suffices to keep a global goal (usually the point where the previous stream allocation ended) for each NUMA node.

2.4.6 Clutser Allocation / Bigalloc

We don't note individual blocks In the allocation bitmap of each group, but groups consisting of blocks and having predetermined size, the so-called clusters [6]. The alternative title results from the `bigalloc` flag that can be used when creating the filesystem. With clusters, we minimize overhead and storage waste due to metadata. In some cases, it can also favor the locality of file's data.

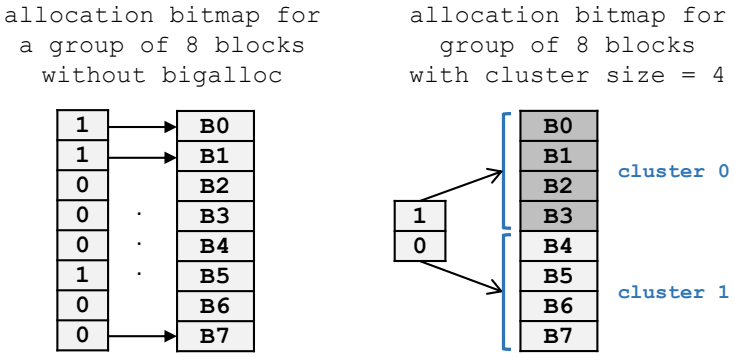

Figure 2.12: Cluster Allocation: Illustration of how the allocation bitmap becomes more compact when increasing the cluster size.

This feature has not been taken into account in our implementation, but its possible effect in terms of NUMA awareness is described later on.

Chapter 3

Methodology

In this section we will cover the majority of the methods used for the purposes of experimentation and kernel development. We will start by introducing in the first two sections the FIO and Filebench tools, which we use to run tests, and mainly for benchmarks. In the following sections, we will look how to configure a suitable virtual machine for the purpose of modifying the kernel, and how to explore and debug the kernel's source code.

3.1 The FIO Tool

FIO is a flexible input-output (I/O) load generation tool for testing and benchmarking. It was written to satisfy the need for systematic testing in the development of Linux's I/O and scheduler subsystem [11]. It offers the ability to configure and execute simple and well defined I/O processes, with detailed statistics collection.

For the purposes of this work, the capabilities of FIO primarily serve to observe the performance characteristics of the non-volatile devices, as we will see in chapter *Experimentation*. At a later moment, FIO also helps with evaluating the correctness and efficiency of the software infrastructure created in chapter *Implementation*.

FIO accepts as a parameter a description of a job, which can spawn multiple system processes, and each process can spawn multiple execution entities (processes or threads) of its corresponding job. Parameterization of the work performed by FIO is done either via the execution parameters of the benchmark utility itself, or through input files that follow specific formatting rules [11]. In table 3.1 we present some of the most basic parameters that we set for the process of collecting our measurements, as seen in files of directory `experiments/dotfio/simple/` of our source code.

Parameter	Function
`thread`	Create workers to perform the requested job via POSIX threads instead of fork.
`group_ reporting`	Print statistics for all execution threads instead of each one individually. Makes parsing the results easier.
`ioengine`	Interface used to perform I/O. For the purposes of our work, we set it equal to "sync," meaning that we use the read and write syscalls. Experimentation with memory mapped files is possible by choosing the "mmap" value for this parameter.
`size`	Set the file size for each thread. For the purposes of our work, it is typically set to 256MB or 4GB.
`bs`	The size of the block (as in the buffer) that we use as a parameter for the individual system calls. We chose the default value of 4KB.
`rw`	Type of I/O pattern. We set it to `randrw` so we can mix writes and reads via the rwmixread parameter. If we want to experiment with the degree of randomness of I/O, we can tune the `percentage_random` parameter.
`rwmixread`	The percentage of memory accesses that are read instead of writes.
`percentage_ random`	Degree of randomness of the I/O. When the parameter is set to 0 we have sequential IO, while for a value equal to 100 we have accesses to completely random parts of the file. For our purposes, we set this parameter equal to 0.
`invalidate`	Invalidate pages (cached in the page cache) of files used by the given job. It only makes sense if DAX is disabled.
`create_ serialize`	If the parameter is equal to 1, the creation of the threads' files is not done by the threads themselves, but by their parent thread, before the creation of its children. Otherwise, each thread creates its own file when it starts executing. Particularly useful parameter when we want to evaluate our ext4 implementation as to whether the data allocation respects the property of NUMA locality.

`loops`	Number of iterations of the given job. Typically equal to 1. Useful as a parameter when we want to test, with DAX disabled, how having cached pages after the first iteration can result in faster throughput of the requested IO.
`numjobs`	How many execution entities will be created for the requested job.
`directory`	The directory in which the files of each execution entity are located or placed. With this parameter, we instruct FIO to use the proper non-volatile memory device, by providing a directory for the corresponding mountpoint.
`cpus_allowed`	List of processor cores we want to restrict the execution of this job to. With this parameter we achieve the execution of the task on a selected NUMA node.

Table 3.1: The most common parameters of FIO used for our purposes

In order to create job execution threads that are distributed among NUMA nodes, we define a separate process for each node. As these processes have many parameters in common, we collect them in `experiments/dotfio/simple/global.fio`. Some differentiating parameters between the processes are shown below the double line of table 3.1. As an example of how these are set, the reader can look at file `experiments/dotfio/simple/mixed.fio`.

As a final note, in `experiments/instances/fio_instance.sh` of our source code, it is shown in detail how to call FIO and set the parameters that we have left variable in files ending with `.fio`. How the `cpus_allowed` parameter is set will be explained in section *Measurement Collection Methodology*.

3.2 The Filebench Tool

Filebench is a benchmark utility for file systems and storage media technologies [28]. It differs from FIO in that the Input/Output traffic it produces can follow more complex patterns, in a way capable of simulating the behavior of various computer applications. This is achieved through its own job description language, called the Workload Model Language.

Filebench's source code is accompanied by several predefined workloads that are used quite a lot in literature [18, 29, 13]. In our work, we will leverage Filebench to evaluate the correctness of our modifications to ext4, presented in chapter *Implementation*, as well as to observe any achieved performance gain.

From the predefined execution scenarios, we utilize those from table 3.2, with some proper adjustments.

Workload	Description
`fileserver`	Simulates the behavior of a file server, which writes new files, appends, deletes, reads or requests information (stat) about existing files.
`varmail`	Simulates the behavior of a mail server user, who deletes, creates, replies to and reads messages. This workload, in its predefined form, considers messages to be 16KB in size on average.
`webserver`	Simulates the behavior of a webserver, which primarily reads requested files, and periodically appends to a log file.

Table 3.2: Filebench workloads used in our work

Note that for our purposes, we are not using Filebench as it is (i.e. in the state of the commit with prefix `22620e6` in the tool's GitHub repository), but we are extending it as needed. Due to this, we created the files in directory `experiments/patches` of our source code. The additions made to Filebench will be described in more detail in section *Performance Assessment via Filebench* of chapter *Evaluation*.

3.3 Measurement Collection Methodology

In order to be able to safely draw conclusions from our measurements, especially in the case of non-volatile memories which have peculiarities as will be seen in chapter *Experimentation*, we need to carry out the measurement collection process in a systematic way. In this section. we are going to see how to reduce the amount of noise introduced in our measurements, and how the measurement collection process was automated to facilitate our workflow.

3.3.1 Limiting unwanted noise in measurements

As previously mentioned, FIO is our primary tool to generate IO to and from the system's non-volatile memories, while logging relevant statistics. Through the input of this utility, we can easily and explicitly specify the number of threads to perform the requested accesses, to which files, how the accessing threads interact with the filesystem, and many other parameters. However, no matter how

precise we are in describing the experiment, we must first minimize the external factors of the system that can add noise to our measurements.

The first and most important point of noise introduction is, naturally, the processor. In more detail, we distinguish immediately afterwards in what ways we may reduce our experiments' unpredictability by minding the intricacies of certain CPU features.

Avoiding Hyperthreading

We believe that hyperthreading can lead to further deviation in measurements. The percentage of time during which threads co-exist on a processing core may vary from one measurement to the next, due to irrelevant threads to our experiment being executed concurrently on the CPU. That is, we prefer when possible that each thread is not placed with another (I/O intensive) thread on the same processing core, but whether this is done in an execution is highly variable. One of the reasons for this preference is related to the assumption that the caches of the processing cores are better and more predictably utilized this way.

Consequently, we utilize at most one logical one for each processing core. This is something we control via the `cpus_allowed` parameter of FIO.

Using only as many cores as necessary

Rescheduling threads to a new processing core burdens data locality in caches. We can limit this phenomenon if we set an appropriate mask for the processors on which the experiment can be executed. In general, we make sure to use a fixed set of processing cores, in number equal to that of the threads for each experiment. We achieve this either with `taskset`, or via the `cpus_allowed` parameter of FIO.

If we wanted to further limit the phenomenon described, we could use parameter `cpus_allowed_policy=split` of FIO, with which we manage to have each thread running throughout its lifetime on a single processing core. Regardless, we note that on one hand such practice makes our experiment even less realistic, and on the other hand, it may lead to underutilization of the system when we have more threads than available processing cores. This is because, if we have for example 17 threads running on 16 cores, then at least two threads will be competing for a core when there may be another one available at some given time. The total time elapsed for the experiment may be almost twice as long compared to having allowed execution on any core of our chosen mask. Also, we get an apparent discontinuity in the performance metrics between 16 and 17 threads.

Setting the CPU's frequency to be static

The dynamic adjustment of the CPU's frequency leads, in an obvious way, to variations between samples of the measurements. To make sure that no noise is introduced due to frequency scaling, we disable this feature. To this end, we use the `cpupower` executable that can be generated in directory `tools/power/cpupower` of the kernel's source code. Additionally, for the sake of certainty, we update `/etc/default/cpufrequtils` accordingly and restart the cpufrequtils service as shown below.

```
echo 'GOVERNER="performance"' | tee /etc/default/cpufrequtils
systemctl restart cpufrequtils

export LD_LIBRARY_PATH="$cpupowerdir"
$cpupowerdir/cpupower frequency-set -g performance
```

Disable the processor's idle states

By the term idle state we refer to a state of a processing core in which its available functions are temporarily limited for the sake of energy efficiency. As in the case of dynamic frequency, we realize that disabling this feature can help towards achieving uniformity between samples, thus enabling more meaningful comparison of measurements. We achieve this in a similar way to that of disabling dynamic frequency:

```
systemctl disable ondemand
systemctl restart cpufrequtils

export LD_LIBRARY_PATH="$cpupowerdir"
$cpupowerdir/cpupower idle-set --disable-by-latency 0
```

In addition, it is important that the experiments are performed in the absence of other users that may utilize significant resources. This is necessary because the intensity of another user's interaction with the system can be random, and unevenly burden the performance of different measurements.

Finally, between repetitions of an experiment we wish to have as identical initial conditions as possible. For example, the file system we use should not have limited capacity due to a previous experiment, because then, during the new iteration of our experiment, the way the storage medium is used may change significantly.

3.3.2 Automated Experiment Execution System

We have mentioned several things to check off before conducting any of our experiments. Executing the experiments manually makes mistakes more likely due to the human factor. Additionally, it would require a significant portion of our time, which we wish to utilize otherwise. For the reasons mentioned, we fully automate the process of running the experiments, collecting the produced data and visualizing the results, with the help of bash scripting. All the necessary files that serve this purpose are located in directory `experiments` of our source code. We briefly mention the subdirectories it contains, and each one's role:

- `dotfio/simple`: This directory contains all the FIO job description files of our experiments. We have multiple `.fio` files because special care is needed in edge cases of thread distribution among NUMA nodes. We have a subdirectory of `dotfio` named `simple`, because we can extend our work to introduce other possible FIO usage scenarios. For example, we can add a "thinktime" directory, whose files may use the homonymous FIO parameter to simulate CPU utilization as if there was processing done upon the accessed data. This way, we would simulate not only I/O, but also CPU intensive threads.

- `graphs/scripts`: In this directory we have python files that can produce many of our final graphs directly. We use the matplotlib and seaborn libraries for this purpose, among others. The resulting images are placed in directory `graphs` .

- `instances`: This directory contains FIO and Filebench parametric execution scripts. The `fio_instance.sh` file, for example, contains the logic of configuring the CPU mask, choosing the right file of `dotfio/simple`, taking multiple samples per measurement, saving the measurements, etc.

- `measurements`: This directory contains some of our measurements, including ones for the NOVA and WineFS filesystems for further investigation, as seen in the appendix.

- `patches`: This directory contains patches with extensions and modifications to Filebench's source code. These will be discussed further in section *Evaluation*.

- `scenarios`: Each file in this directory contains a complete description of each experiment presented in this document. Said files take advantage of what is available in all the other subdirectories listed, automating the entire process, from measurement collection to producing the figures. Based on these files, we can write new experiments with minimal effort.

- `scripts`: Here we have collected all the bash scripts needed for our experiments. Briefly, some examples include subdirectory `freq` with scripts taking the necessary steps to reduce noise due to the CPU, as well as subdirectory `mount` with files related to mounting devices and creating file systems on them.

If the reader wishes to take advantage of this infrastructure, we note that it is necessary to use the Bash shell due to how our system was implemented. Finally, the `create_serialize` parameter of `experiments/dotfio/simple/global.fio` needs special attention, as it was left to be set manually. For the purposes of chapter *Experimentation*, we prefer a value of 1, since it reduces the noise in our measurements when creating the files, due to concurrency at the filesystem level. In chapter *Evaluation* however, and specifically in section *Correctness Assessment*, we will see that in order to make the files' data allocation local for each thread, we must set `create_serialize` equal to 0.

3.4 Virtual Machine Configuration

For modifying the Linux kernel and subsequently evaluating the correctness of our changes, we leverage the QEMU (Quick Emulator) emulator in combination with the KVM (Kernel-based Virtual Machine) virtualization module. The emulator is what allows us to execute and specify the characteristics of our virtual machine for the purposes of kernel development. The KVM module allows the host's kernel to function as a hypervisor managing QEMU's virtual machine, and by extension it allows the utilization of possible hardware virtualization extensions of the underlying processor.

3.4.1 Defining the Characteristics of our Virtual Machine

Our first step in establishing the required kernel development infrastructure is to configure a virtual machine that 1) emulates a NUMA system, describing the devices attached to each node, and 2) emulates non-volatile memory devices.

For (1), it suffices to define NUMA nodes and attached volatile memory devices as parameters of QEMU's, in a manner similar to the following:

```
-object memory-backend-ram,size=2G,id=m0
-object memory-backend-ram,size=2G,id=m1

-smp 4,sockets=2,maxcpus=4
-numa node,cpus=0-1,memdev=m0,nodeid=0
-numa node,cpus=2-3,memdev=m1,nodeid=1
```

In defining the non-volatile memory devices afterwards, we must specify the node to which we attach each device, via the `node` parameter.

For (2), we initially enable support for virtual non-volatile devices on QEMU's side:

```
-machine pc,nvdimm=on
-enable-kvm
```

Next, we define each device as shown in the following example:

```
-object memory-backend-file,id=nvm0,share=on,
    ↪ mem-path=nvdimm/nvdimm0,size=4G
-device nvdimm,id=nvdimm0,memdev=nvm0,node=0
```

We specify a backend file that is created on the host, providing its directory and its size. Then, we create an NVDIMM device based on this backend file. Option `share=on` ensures that the backend file is shared between host and guest, so the configuration of virtual non-volatile devices (as well as the contained files) will remain unchanged when restarting the virtual machine.

We make one device for each node. Ideally, we would like to have two or more devices for each node, so we can experiment with creating an interleaved namespace on each. However, the way this is achieved, i.e. creating a single memory allocation goal per node as shown at the end of [15], is through intel's IPMCTL tool. This is in turn dependent on the devices' firmware. Obviously, at the level of a virtualized non-volatile device, it is not possible to successfully execute the tool for the aforementioned purpose. Therefore, we are practically limited to a single device per node when in a virtualized environment.

What we have mentioned so far covers the most interesting parts of configuring the virtual machine. More details, such as the parameters for initially installing a Linux distribution and providing networking to the VM, can be seen at later subsections, as well as in `various/vm.sh` of our source code.

Finally, we note that our purpose with the VM configuration is not to create a virtual machine that qualitatively approaches the performance characteristics of a similar real system. For example, we are not interested in simulating interconnection network latency (NUMA effect), nor in implementing any methodology for approximating the bandwidth of non-volatile devices by rate-limiting accesses to host's DRAM (as was usual in research before Intel's non-volatile devices became commercially available [31]).

Our goal is to have a virtual system that will be convincing enough for the kernel to act approximately the same as in our real machine. We exclude of course the operation of drivers virtualizing devices that are missing from the VM. Thus, we can determine with heuristic methods applied within the VM, what part

of Linux's codebase is relevant to the DAX feature or the (more abstract part of the) NVDIMM code, gaining the understanding necessary to modify it properly and check if our modifications function correctly (but not necessarily see where we stand performance-wise).

3.4.2 Installing a Linux Distribution

Debian Linux was the distribution of choice. We download the needed ISO file for the installation, and create the qcow2 formatted virtual storage via the following command:

```
$ qemu-img create -f qcow2 debian.img 25G
```

Before we boot the virtual machine for the first time, we make the following two parameters of QEMU available so that we can boot from an optical media filesystem (ISO image) and proceed with the installation:

```
-cdrom <ISO-image> # In this case: debian-11.5.0-amd64-DVD-1.iso
-boot d
```

We run the virtual machine script and follow the procedure of a standard installation of the selected Linux distribution. The only needed deviation from the default settings during this process is to bypass the installation of a graphical user interface (in this case, the default option was GNOME). This is desirable because including a GUI adds significant overhead to the installation itself, and it will delay the startup of the virtual machine. Instead, we enable option "SSH server" so that we can connect to the virtual machine from a terminal emulator on the host. This is preferable to using e.g. the graphical interface that QEMU provides (the GTK display) for interacting with the virtual machine, since it has limited capabilities.

After the installation is complete, we remove the virtual machine script parameters related to installing the selected distribution, and we start the machine to configure our new installation. We log in to the virtual machine as the root user. The steps we follow to complete the virtual machine setup are roughly the following:

```
$ nano /etc/apt/sources.list # delete the cd repository if still
    ↪ there (seems like a debian 11 problem)
$ apt-get update
$ apt-get upgrade
$ apt-get install sudo vim git make gcc flex bison
    ↪ libncurses-dev libssl-dev libelf-dev bc ndctl
    ↪ original-awk
```

```
$ /sbin/adduser <non-sudo-user> sudo
$ vim /etc/ssh/sshd_config # add the line "PermitRootLogin yes"
$ systemctl restart sshd
```

In summary, we install that packages necessary to compile the Linux kernel and run the source code of this work, and we configure SSH to allow logging in as root. The latter is potentially quite a bad security practice, but it greatly facilitates the process of developing the kernel with the aid of the virtual machine: the main reason here is that we avoid the sudo interface when we want to modify and diagnose the system under our experimental kernel, mostly to save time and effort. We now connect to the virtual machine as follows:

```
$ TERM=xterm ssh -p 2222 <root or user>@127.0.0.1
```

Next, we configure the virtual non-volatile memories by creating the corresponding namespaces in fsdax mode:

```
$ ndctl create-namespace -f -e namespace0.0 --mode=fsdax
$ ndctl create-namespace -f -e namespace1.0 --mode=fsdax
```

At this point the virtual machine is ready for doing kernel development. As mentioned, it adequately simulates a NUMA system with attached virtual non-volatile memories, which is satisfactory for our purposes. However, the development of the userspace component encounters limitations in the virtual machine environment, as we cannot test and develop upon certain features that cannot be virtualized, i.e. reading the hardware counters of the devices with Intel's IPMCTL API. So, we will limit ourselves to reading values from the procfs interface, and testing features unsupported in our VM will necessarily be done on a machine with real non-volatile memories.

3.4.3 Compilation and Installation of the Kernel

Note that the last version of GCC the gives us a successful compilation is the 11th. From version 12 onward there is incompatibility with the implementation of `xrealloc` for the selected kernel version. Compiling and installing the kernel developed for the purposes of this thesis can be done as follows:

```
$ git clone https://github.com/PhTof/Diploma.git
$ cd Diploma/linux-5.13
$ cp ../various/minconfig_5.13 .config
$ make oldconfig
$ make -j4 && ./aftermake.sh && reboot
```

We have two things to notice here: the first is the prepared configuration `minconfig_5.13`, and the second is script `aftermake.sh`, both of which we analyze later on.

Minimal Configuration

Available with our source code is a kernel configuration for the virtual machine, which is intended to be as limited as possible but complete in terms of supporting non-volatile memory modules and the DAX feature. The basic rules in producing the configuration from the default one are 1) the removal of drivers related to many network and other devices (external USB devices, graphic cards, etc.), which certainly do not have any use in our virtual machine, and 2) adding any configuration parameters that had an identifier including the terms "DAX," "PMEM," or "NVDIMM."

Table 3.3 lists in more detail the most important options that we need to change in the default configuration. The reasoning behind removing several unused drivers is for achieving faster recompilation of the kernel (which action will be too frequent, as we need to quickly test our modifications), given that the compilation speed is significantly impaired when we compile multiple unnecessary modules.

Kernel Version
[CONFIG_LOCALVERSION] General Setup > Local version - append to kernel release : Add an identifier for the kernel.

Support non-volatile memory as storage
[CONFIG_ACPI_NFIT] Power management and ACPI options > ACPI (Advanced Configuration and Power Interface) Support > ACPI NVDIMM Firmware Interface Table (NFIT) : Identify non-volatile devices at boot.
[FS_DAX] File systems > Direct Access (DAX) support : Direct access (DAX) support.
[MEMORY_HOTPLUG] Memory Management options > Allow for memory hot-add : Dependency of [ZONE_DEVICE].
[MEMORY_HOTREMOVE] Memory Management options > Allow for memory hot remove : Dependency of [ZONE_DEVICE].
[ZONE_DEVICE] Memory Management options > Device memory (pmem, HMM, etc...) hotplug support : Provides the ability to create namespaces in fsdax mode.

[LIBNVDIMM] Device Drivers > NVDIMM (Non-Volatile Memory Device) Support : More general non-volatile device support, automatically selected by [CONFIG_ACPI_NFIT].

Settings related to NUMA topology

[ACPI_HMAT] Power management and ACPI options > ACPI (Advanced Configuration and Power Interface) Support > NUMA support : Recognition of NUMA topology at startup.

Other settings related to non-volatile memories

[X86_PMEM_LEGACY] Processor type and features > Support non-standard NVDIMMs and ADR protected memory : Support non-standard NVDIMMs.

[TRANSPARENT_HUGEPAGE] Memory Management options > Transparent Hugepage Support : Dependency of [CONFIG_DEV_DAX].

[CONFIG_DEV_DAX] Device Drivers > DAX: direct access to differentiated memory > Device DAX: direct access mapping device : Namespace configuration in devdax mode.

[CONFIG_DEV_DAX_KMEM] Device Drivers > DAX: direct access to differentiated memory > KMEM DAX: volatile-use of persistent memory : Use of non-volatile memories as RAM.

[CONFIG_DEV_DAX_PMEM] Device Drivers > DAX: direct access to differentiated memory > Device DAX: direct access mapping device : Raw access to non-volatile memory.

[VIRTIO_PMEM] Device Drivers > Virtio drivers > Support for virtio pmem driver : Support for virtio-pmem devices.

Settings for networking

[CONFIG_ETHERNET] Device Drivers > Network device support > Ethernet driver support : Disable all but Intel devices > Intel(R) PRO/1000 support [CONFIG_E1000] [CONFIG_ACPI_PCI_SLOT] Power management and ACPI options > ACPI (Advanced Configuration and Power Interface) Support > PCI slot detection driver : Otherwise we have interface enp0s3 instead of ens3, which is configured during the installation of the selected Linux distribution (PCI device vs. hotplug device). If we end up with an interface other than the one configured during installation, we need to run command `dhclient -d <interface>`.

Alternative settings for networking

[CONFIG_VIRTIO_CONSOLE] Device Drivers > Character devices > Virtio console : Dependency of [VIRTIO].

[VIRTIO_NET] Device Drivers > Network device support > Virtio network driver : VIRTIO network interface support. It requires some alternative parameters in QEMU, but it provides a faster network interface.

Settings to minimize the configuration

[DRM_I915] Device Drivers > Graphics support : We don't need graphics support since we will be connecting via SSH.

[CONFIG_SURFACE_PLATFORMS] Device Drivers > Microsoft Surface Platform-Specific Device Drivers : We don't need vendor specific drivers.

[CONFIG_MACINTOSH_DRIVERS] Device Drivers > Macintosh device drivers : We don't need vendor specific drivers.

[CONFIG_SOUND] Device Drivers > Sound card support : We don't need sound card support.

[CONFIG_USB_SUPPORT] Device Drivers > USB support : We don't need USB support, unless USB passthrough facilitates file transfer. But we can do this to some extent through the `scp` command.

[CONFIG_WLAN] Device Drivers > Network device support > Wireless LAN : We have network support via a virtual ethernet interface, so we don't need wireless networking.

[CONFIG_WIRELESS] Networking support > Wireless : Further disable wireless networking.

[CONFIG_NETWORK_FILESYSTEMS] File systems > Network File Systems : It is not necessary to use network filesystems.

[CONFIG_MISC_FILESYSTEMS] File systems > Miscellaneous filesystems : No filesystems from this category was needed.

Settings for kernel debugging capabilities

[CONFIG_DEBUG_INFO] Kernel hacking > Compile-time checks and compiler options : Kernel debugging support.

[CONFIG_GDB_SCRIPTS] Kernel hacking > Compile-time checks and compiler options > Compile the kernel with debug info > Provide GDB scripts for kernel debugging : Better debugging support through GDB.

Table 3.3: Options modified for our kernel configuration

The options regarding non-volatile memories are equally important to check if enabled even in the evaluation system, as, for example, we cannot be certain that the adaptation of an older kernel configuration (command `make oldconfig`) will properly set the attribute `FS_DAX`.

Additionally, between installations of different kernel versions, we need to make sure that the devices' namespaces are left intact, to reset it to the wanted setting if needed. In particular, if we install a kernel for which some option regarding the DAX feature is not set correctly or it does not exist, then the namespaces we have defined will go from the desired fsdax mode to raw. Even after installing a properly configured kernel, the mode transition will not necessarily be undone. It is therefore important to do a check with the aid of command `ndctl list -N`, or with command `journalctl -b 0 -k` after trying to mount a filesystem with DAX enabled. In the latter case, a related message of the DAX attribute not being supported will appear if something is not configured correctly.

Aftermake Script

The kernel compilation process is not the only time bottleneck, since during virtual machine startup, reading a large initrd (initial ramdisk) file can lead to long waiting times. This can prove time-consuming in cases where we need to test the functionality of some small change in the kernel code (perhaps printing some diagnostic message via function `printk`). To limit the effect of this factor, we take care to remove any unneeded symbols from the modules before generating the initrd and after compiling everything. This is achieved with the following command:

```
$ find /lib/modules/<kernel_ver><local_ver>/ -name *.ko -exec
   ↪ strip --strip-unneeded {} +
```

This leads to a drastic change in the size of the initrd file, and consequently a significant reduction in the boot time of the virtual machine. Perhaps this may not a good practice, or there exists some option available in the kernel configuration that allows the removal of unnecessary symbols through its compilation infrastructure. But for practical purposes, it has proved useful.

As a first note we mention that when starting the virtual machine with the modified kernel, it is possible to get a message stating `"wrong EFI signature"`. This may be explained by the fact that QEMU itself does not use any available UEFI firmware [23] without additional parameters. We can remove UEFI support from the kernel with some configuration changes, but it is not necessary for the purposes of our work.

Finally, we note that compiling our modified kernel using the default configuration of kernel version 5.13 is likely to result in compilation errors. This is due to the lack of alternative definitions of the newly introduced functions/symbols in `#if[n]def … #else … #endif` sequences. There is no point in compiling this kernel if support for non-volatile devices is not available via the configuration, so the codebase is left as is. Reference to this fact is made to cover the case in which the reader might attempt to compile the kernel without first providing a compatible configuration.

3.5 Tracing in the kernel

For any modification we want to make to the kernel codebase, we first need to locate its relevant functions that implement what we want to change. For example, as we will see in section *Implementation*, in order to properly modify the ext4 filesystem, we first want to locate the functions related to (meta)data allocation. Although the execution path in each case can be quite complex and involve many individual auxiliary functions' invocations, we are content for now with just qualitative identification of the candidate functions.

Additionally, there can be many subcases that greatly affect which functions are executed and how. Going back to the example, ext4 may have files configured not to use an extent tree to index their data blocks (by disabling the `EXT4_INODE_EXTENT` flag). However we will limit ourselves to the most common case, i.e. what we can observe by tracing the called functions in a series of simple experiments of creating new files (i.e. inodes) and writing to them via FIO or Filebench.

3.5.1 Tracing with ftrace

The tracing of the called functions in the kernel is achieved through the following command of `perf` (which in this case works as a simple wrapper of kernel's ftrace [27] utility):

```
$ perf ftrace -a --inherit -T "<function_name>"
    ↪ <experiment_script>
```

The important thing is that the value of the `<function_name>` field can even be a simple regex. In particular, the use of the asterisk (*) operator is allowed, and even more than once. Therefore, if we want, for example, to record via perf the functions related to multiblock allocation, we set `<function_name>` = `ext4_mb_*`, while if we simply want to record every call relevant to ext4, we can set `<function_name>` = `*ext4_*`.

Note that we may need to configure and compile the kernel appropriately to be able to use ftrace, or even to allow tracing in certain parts of the kernel's code.

If we want to record via `perf` the sequence of calls that lead to a function we are interested in, we can use the following logic:

```
#!/bin/bash

perf=$(echo $HOME/linux-5.13/tools/perf/perf)
func=submit_bio
executable=./simple

prepare_state() {
./simple # Be sure the file exists
}

restore_state() {
: # Nothing to be done here
}

get_caller() {
        func=$1
        # tail -1: keep only the last call of the desired function
        # awk '{print $NF}' | cut -c3- : output the caller
        $perf ftrace -T "$func" $executable  | tail -1 | \
                awk '{print $NF}' | cut -c3-
}

prepare_state

while : ; do
        # Print the current function
        echo $func
        restore_state
        func=$(get_caller $func)
        if [[ -z "$func" || "$func" == "do_syscall_64" ]]; then
                break
        fi
done
```

Executable `./simple` in our example simply writes to a file. We could achieve something similar via command `bash -c "echo ... > file"`. However, this should be avoided as a practice, because the redirection operator has idiosyncratic behavior. We will expand more on this in section *Correctness*

Assessment of chapter *Evaluation*. Additionally, it is desirable to limit as much as possible the operation of the provided executable to only the absolutely desired, to limit tracing complexity. In particular, writing the file with some bash command would perform many additional operations related to the implementation of bash itself, rather than just the process of writing the file. The functions `prepare_state` and `restore_state` are intended to condition the same sequence of calls to occur each time.

Note that during kernel compilation many functions called from only one point tend to be inlined into the single calling function for optimization purposes. But this way the function will not include them in the list of traceable functions (`/sys/kernel/tracing/available_filter_functions`). In this case, we need to add the noinline symbol to the beginning of the function definition we want to register, and recompile the kernel. This is time consuming, and our purposes are probably better served by one of the next methods.

The simplest and most commonly used method of finding inlined functions is to record the call sequence, and find the missing intermediate steps by reading the code through the Elixir Bootlin online tool. This tool has performed a static analysis of the kernel symbols, and can provide, for each one, possible definitions and references in the codebase of the chosen kernel version. So, we can follow the sequence of traced functions by reading the relevant source code, and, when intermediate functions are missing, follow called functions with a single reference to see if we somehow end up at the next function in the sequence. This process can prove time-consuming if we have many intermediate, inlined functions, but in the average case we can relatively quickly locate them.

3.5.2 Using the dump_stack() kernel function

If we want to get a detailed and accurate call sequence for a particular function, we place a call to the `dump_stack()` function at whatever point we want. This practice requires recompiling the kernel, and if `dump_stack()` is placed in a frequently called function, we will have a significant slowdown in the execution of the function due to system journaling.

3.5.3 Creating a kernel debugging infrastructure via QEMU

The best way to get a full picture of how the kernel works is to debug it, as we would any simple executable. Obviously this isn't that simple, since the way an operating system runs is very complex, and it can't easily run a program that controls the execution flow of the operating system itself. Fortunately there is infrastructure for debugging kernel code via QEMU in conjunction with GDB (GNU Debugger).

Our first step is to compile the kernel. We can do this either directly on the host, or in the virtual machine. It is important in any case to make sure that the generated modules are installed on the virtual machine system (as would result from command `make modules_install`), and to embed in the kernel anything we want to test via debugging. For example, in our case we make sure to choose, through command `make menuconfig`, the ext4 filesystem to be built into the kernel (`<*>`) rather than compiled as a module (`<M>`).

There are three necessary files for debugging, which are listed below. In the case of compiling the kernel inside the virtual machine, we need to copy them after compiling to the host system. This practice was preferred as it facilitates the creation of the files. All the commands mentioned are executed in the directory containing the kernel source code.

- `initrd`: The initial ramdisk is necessary so that, in combination with the kernel image `vmlinuz`, QEMU can be started directly, without the intervention of a bootloader. It is produced via command `make install` in directory `/boot/`, as a file of the form `initrd.img-<kernel-ver><local-ver>`, where `<kernel-ver>` is the version of the kernel (here 5.13.0) and `<local-ver>` is the name we gave the kernel via parameter `CONFIG_LOCALVERSION` of the configuration. Before the `make install` command, the kernel must be compiled (`make -j`nproc``) and the modules must be installed (`make modules_install`).

- `vmlinuz`: vmlinuz is the compressed kernel executable without debug symbols. It is the file `<kernel-dir>/arch/x86/boot/bzImage` resulting from the kernel compilation, where `<kernel-dir>` is the directory of the compiled kernel source code. Equivalently, on a Debian distribution this file is located in directory `/boot` after the kernel is installed, and can be of the form `vmlinuz-<kernel-ver><local-ver>`.

- `vmlinux`: vmlinux is the kernel executable with debug symbols, given that these have been requested via the configuration process (option `CONFIG_DEBUG_INFO`). It is produced with command `make vmlinux`, and file `vmlinux` ends up in the directory containing the kernel source code.

The first two files are used as QEMU parameters, and allow enabling the remote debugging option via GDB on the host. The third serves as a parameter to GDB so that it can leverage its symbol table. In summary, in the virtual machine we run the following commands:

```
$ cd <kernel-dir>
$ make -j`nproc`
$ make modules_install
$ make install
$ make vmlinux
```

Having done this preparation, we transfer the required files to the host with the following commands, where <QEMU-dir> is the directory on the host where we have the QEMU scripts. Then we turn off the virtual machine. We assume that <ver> = <kernel-ver><local-ver>.

```
$ scp -P 2222 root@127.0.0.1:/boot/vmlinuz-<ver> <QEMU-dir>
$ scp -P 2222 root@127.0.0.1:/boot/initrd.img-<ver> <QEMU-dir>
$ scp -P 2222 root@127.0.0.1:<kernel-dir>/vmlinux <QEMU-dir>
```

Having collected the necessary files, we update the QEMU script by adding the following parameters:

- `-kernel vmlinuz-<ver>`: The kernel image we will use for booting.
- `-initrd initrd.img-<ver>`: The initial ramdisk that the virtual machine will use to boot.
- `-append "root=/dev/sda1 nokaslr console=ttyS0"`: We determine, again for the boot process, where the base (root) of the file system of the Linux installation is located in the virtual machine (in this case `/dev/sda`). We disable kernel address space layout randomization (`nokaslr`) so that there is a stable correspondence between vmlinux addresses and those in the virtual machine. Finally, for convenience we want the QEMU output to be printed to the terminal from which the script is being run, instead of a separate window. This is what the `console=ttyS0` parameter in combination with QEMU's `-nographic` option allows us to do.
- `-s -S`: The `-s` option enables kernel debugging via QEMU, while the `-S` option delays starting the virtual machine until we continue the execution after connecting with GDB.

We start the virtual machine, and prepare GDB on the host as follows:

```
$ gdb <QEMU-dir>/vmlinux
(gdb) target remote :1234
```

At this point we do whatever preparation we want (e.g. breakpoint definition) and start the virtual machine by providing GDB with command `c`. We can

stop the execution of the virtual machine at any time by pressing `Ctrl + C` in the GDB interface. From now on we have available all the capabilities that the debugger provides.

Some useful commands are the following: `b <function name>` to set a breakpoint, `backtrace` to print the sequence of function calls that led to the function at which execution stopped, and `print <variable name>` to print the contents of a variable of interest (we can choose from the functions in the sequence via command `f <frame number>`).

This method is very powerful, and has a double role since it serves both to record the sequence of functions that leads us to some function of interest, and to debug the modified kernel. But it involves a time-consuming, complex process of transferring and utilizing the necessary files, so we tend to resort to it when other methods are not fruitful.

Finally, the `grep` tool is particularly useful in reading the kernel source code. Having the kernel code available locally, the following command can give us valuable information on finding a function, the definition of a struct, etc.:

```
$ grep -Irn "<pattern>" <kernel_dir>
```

In the continuation of this document, whenever we describe the results of investigating the kernel's source code, we will avoid explicitly mentioning which methods from the above was used.

Chapter 4

Experimentation

In this section we will present the results and findings from a series of experiments to study the behavior of non-volatile memory modules while performing multi-threaded I/O. The key tool that aids our experimentation is FIO. We focus mainly on how the performance of non-volatile memories degrades when there are threads performing remote accesses, i.e. they send requests to non-volatile memory modules of some non-local NUMA node. The resulted measurements lead to the conclusion that there exists a fundamental performance delicacy of non-volatile memories when they are accessed remotely, which seems to be triggered by a very basic pattern of accesses.

In the next chapter, we will see how the vulnerability we mentioned contributes to the formation of our work's motivation. Additionally, more information on how we can further investigate the phenomenon observed here, is provided in chapter *Appendix*.

4.1 Experimental Setup

Next, we present some technical characteristics of the system we used as part of our evaluation infrastructure. We retrieve the information presented afterwards through the following commands:

```
$ lscpu
$ sudo ipmctl show -dimm
$ sudo ipmctl show -a -dimm 0x0000
$ sudo dmidecode -t 17
```

The system's most important parameters are the existence of two NUMA nodes and having installed three non-volatile memory modules per node. Thus, we expect to have per node a cumulative bandwidth equal to roughly 19.8 GB/s

for reads and 6.9 GB/s for writes.

General Characteristics	
Kernel Version	5.13.0
GNU/Linux	Ubuntu 18.04.6 LTS distribution
Processor	
Model	Intel(R) Xeon(R) Gold 5218T CPU @ 2.10GHz
Number of processing cores	32
Threads per processing core	2
Number of NUMA nodes	2
CPU Frequency Range	1.0 GHz – 3.8 Ghz
Caches	L1d: 32K, L1i: 32K L2: 1024K L3: 22528K
Non-volatile Memory Modules	
Model	NMA1XXD128GPS
Capacity per module	126.4 GiB
Number of modules per NUMA node	3
Firmware version	01.02.00.5355 or 01.02.00.5446
RAM Modules	
Model	36ASF4G72PZ-2G9E2
Capacity per module	32 GiB
Number of modules per NUMA node	3
Timing	2666 MT/s

Table 4.1: Characteristics of our evaluation system

We focus on hyperthreading being supported, as shown by the fact that we have 2 threads per processing core, so 64 logical cores.

4.2 Experiments using FIO

In this section, we will attempt to examine the behavior of non-volatile memory modules with respect to how they are arranged in a NUMA topology.

4.2.1 Fundamental Performance Characteristics of the NVM modules

The first experiment we conduct aims to examine the fundamental characteristics of local and remote accesses to non-volatile memories. We use ext4 in its unmodified form (of kernel version 5.13) as the underlying filesystem for device `/dev/pmem0`, which corresponds to the interleaving of all the non-volatile devices on NUMA node 0.

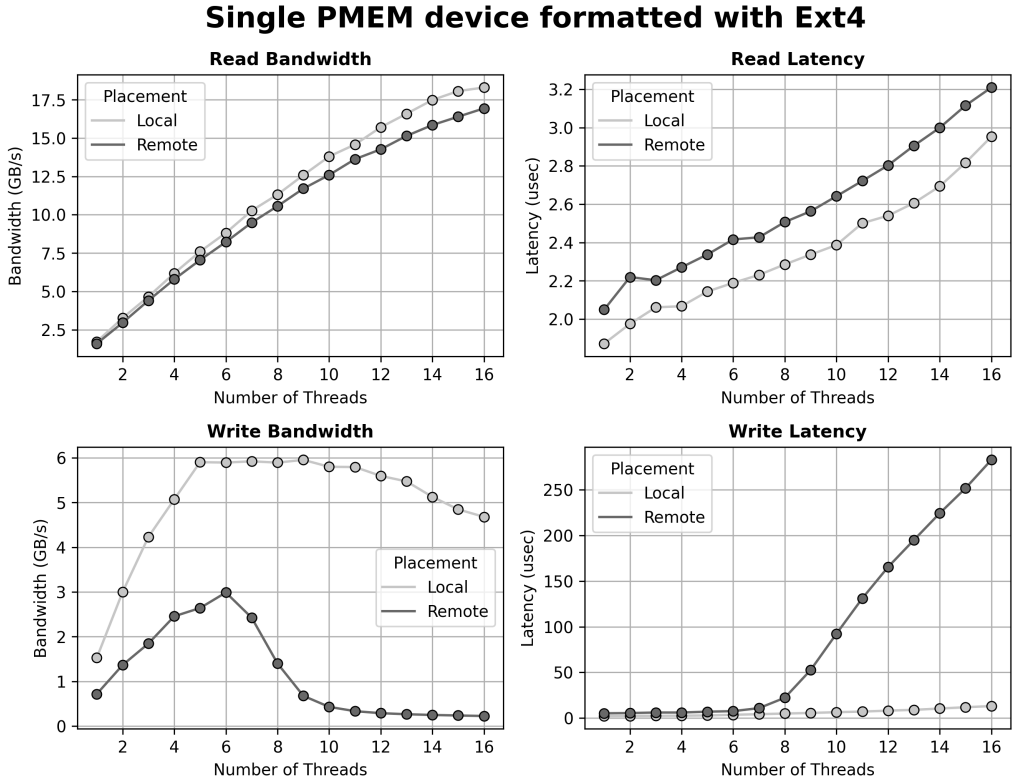

Figure 4.1: Fundamental characteristics of non-volatile memories (using the last sample of each measurement)

We consider two distinct ways of placing threads: "local placement," for which all threads are pinned on NUMA node 0 (where the NVDIMMs of the used device are installed), and "remote placement," having all threads on NUMA node 1 instead. Each thread works on its own separate file, and accesses are either read-only or write-only. Files are accessed sequentially. We take three samples per measurement, and keep the most recent one. Additional details on the experiment are available in `experiments/scenarios/primitive.sh`.

Note that in order to reduce noise and thus smooth out our curves, we used files of at least 1 GB size, in this case 4 GB. The metrics we look at are "throughput" and "latency," while the variable of the horizontal axis is the number of threads executing concurrently.

In terms of read performance, we observe that throughput doesn't seem to be that much different for each thread placement configuration. As expected, there is some elementary delay added to latency in case of remote read accesses, due to the interconnection network bridging NUMA nodes.

In terms of write performance, the results are more complex. First, we notice that even for local accesses, the bandwidth of non-volatile memories is exhausted quickly, and after a certain threshold it even starts to decrease. For remote access, the situation is much worse. We have at best half the bandwidth utilization compared to local accesses, but also very steep performance degradation as the number of threads increases. We also notice the latency increasing rapidly.

The immediate conclusion we draw from this figure is that it is advisable to avoid remote writes to preserve device performance. This is of course something that has been previously documented in literature [31].

4.2.2 The "Read After Remote Write" Effect

If we look more closely at the samples we obtained from our previous experiment, we notice that there is a non-negligible variation between the samples for each measurement. In particular, allowing for all the samples to be plotted, by forming each point corresponding to an experiment from the mean of its samples, while outlining the variance of each measurement, we get the result shown in figure 4.2.

For write accesses there is no big variation, but remote reads show high variability and noticeably worse performance than we expected based on figure 4.1. We focus on bandwidth. The first sample of each measurement, corresponding to the lower limit of the sketched area, seems to follow the behavior of remote writes. The second sample, as roughly indicated by the average, performs slightly better, but is still quite worse than we would expect. The third measurement, corresponding to the upper limit of the sketched area, is the one representative of the expected performance.

After a related investigation, we realized that the reason for the observed phenomena is the fact that we deleted FIO files in order to create new ones before each measurement (following the principle of always having the same initial state for each experiment). Creating the file requires writing to it. Therefore, the first sample has the worst performance and is the one that first reads after writing to the corresponding file. What's notable here is that the second sample is also affected, even though it doesn't directly follow having written to the file, like

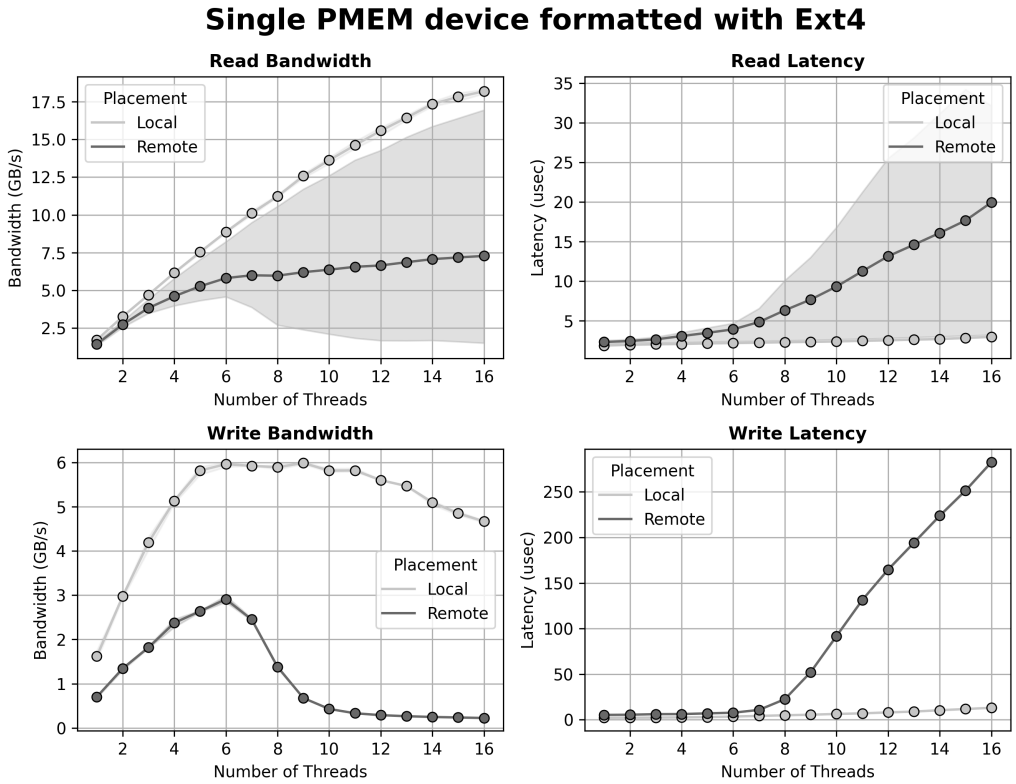

Figure 4.2: Fundamental performance characteristics of non-volatile memories (all samples)

in the case of the first sample. Only with the third sample do we get something intuitively acceptable.

We will henceforth identify this phenomenon as "Read After Remote Write." Stated simply, reads following a remote write have significantly reduced performance. The reason why this happens was not possible to fully investigate within our work's time frame. Despite that, some relevant investigation conducted is listed in section *Investigation of the "Read After Remote Write" Effect* of the appendix.

So we understand that it's not just remote writes that are problematic, but under certain circumstances all remote accesses may experience reduced performance. In fact, mixed accesses, i.e. reading and writing to a file, is not at all a rare occurrence. This is why, in the next subsection, we examine what happens in such a scenario.

4.2.3 Behavior of mixed accesses

For our second experiment, we want to examine what happens if, in addition to pure writes and pure reads, we have a case in which half the accesses to the file are reads, and the other half are writes. This is set directly via the rwmix parameter of FIO.

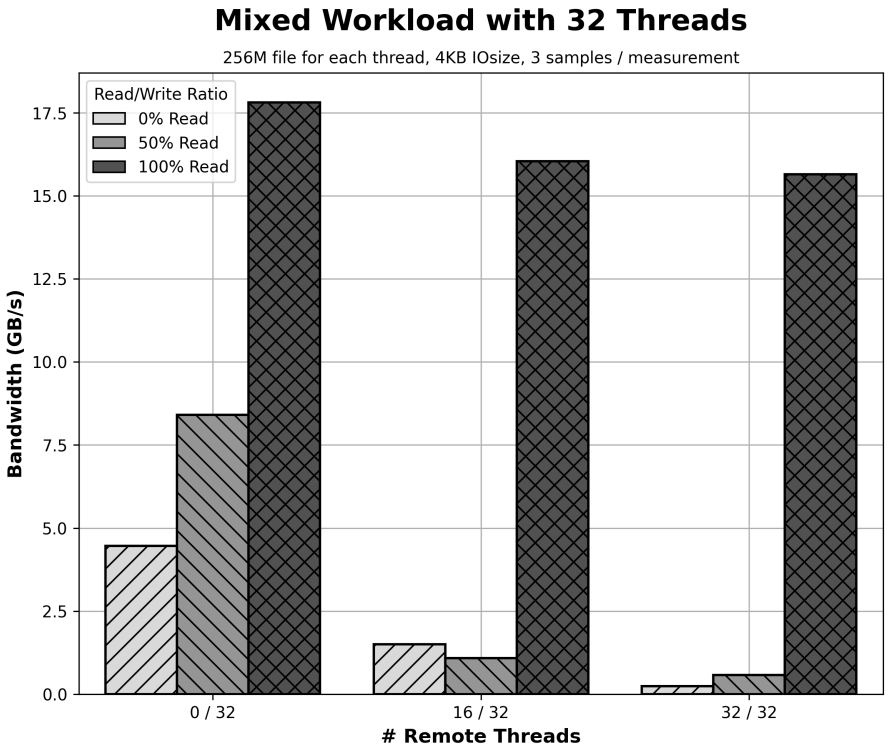

Figure 4.3: Behavior of non-volatile memories for mixed accesses and mixed thread distribution

Until now, the allocation policy has been for all threads to be on the same node, so we had a clear distinction of local and remote placement. We would also like to examine how different the resulting behavior would be if we allow another scenario, where some threads are on one node and the rest are on the other. Recall that we are leveraging the non-volatile devices of only one node, so one class of threads will do local accesses, while the other will do remote accesses, and we allow these two classes to coexist in the same execution. We assume that we have a fixed number of threads, equal to 32, and vary their distribution among the nodes. For each measurement we take three samples, and keep the best one.

So we get the result shown in figure 4.3. Different shades correspond to the

percentage of accesses being reads, while the horizontal axis indicates how many of the 32 threads are remote.

We observe the following:

1. For the case where all threads are local, the numbers are relatively predictable. The case of mixed accesses is not exactly the linear combination of pure reads and writes, but it behaves within logical bounds.

2. For the case where all threads are remote, we again see the dramatic difference between writes and reads, and indeed mixed accesses have a cumulative bandwidth very close to that of writes. This in itself highlights how harmful the "Read After Remote Write" effect really is.

3. In the case of equal distribution of threads between nodes, the picture we get is closer to the case of purely remote accesses than to that of purely local accesses. Besides, here not only are mixed accesses overburdened, but they are also slightly worse than pure writes.

We understand that, due to the "Read After Remote Write" phenomenon, it is of utmost importance for the overall performance of the system that any form of remote access is avoided whenever possible.

Chapter 5

Motivation

Compared to other storage media, non-volatile memories provide a significant comparative advantage based on our performance metrics. At the same time, however, their technology doesn't allow them to have the same capacity per device compared to more traditional storage media, and we can attach a limited number of devices per node. This fact, combined with the need to keep up with the scaling that the NUMA architecture seeks to achieve, leads us to consider a logical interface that will efficiently unify all of the system's non-volatile devices, regardless of NUMA node.

The mentioned interface will make available the combined capacity of the devices under a single level of abstraction, more specifically a customized filesystem that by construction will avoid as much as possible the pathologies presented in chapter *4*. Various proposals could be mentioned in this direction, among which moving data between memories of different nodes, or even conditionally involving the page cache to minimize remote accesses to the non-volatile devices. In fact, the latter idea has not been extensively explored in the literature so far.

Basic design principle: To maintain design simplicity and efficiency, we seek to favor the simplest case: we want to ensure that allocations are made to underlying non-valatile memory modules that belong to the same NUMA node as the accessing process. Admittedly, this property alone does not prove particularly useful in situations where already existing files of other nodes are being used, at least not when these files are not extended. However, we rely on the assumption that a process creates part of its working fileset during its lifetime, or often only extends some already existing fileset. If the process is not rescheduled to different nodes too often, it is clear that the local allocation property will potentially increase the percentage of local accesses as well, yielding performance gain.

Select File System. Having formulated our main goal, we move on to the selection of a suitable file system. At the time of writing, there are 2 research

filesystems available with design choices tailored to specific properties of non-volatile memories. These filesystems are nominally NOVA [30] and WineFS [18]. However, neither has widespread adoption, and they only allow direct access (DAX) operation on devices. This fact may introduce difficulties in the future if we decide to extend this work to conditionally include the page cache in the data path between non-volatile memories and processes.

Ext4 is considered a suitable candidate filesystem. Although it falls short in that it is not specifically designed to work with non-volatile memories, it is widely used and has support for setting the DAX attribute on a per-file basis. In particular, since version 5.13, the only condition for switching file access mode for a file is it not being open by processes. In previous versions, there were other conditions, which made it practically impossible to dynamically set the feature [8] [9]. This allows much more easily the aforementioned possible extension.

Mechanism scheduling threads to NUMA nodes: Having implemented the property of NUMA-local allocations in ext4, it is a logical consequence to wish for a mechanism that will examine parameters of the system state and choose each time the best NUMA node to place each executing thread to. Parameters reported could include how pressured non-volatile memory are (total bytes from read and write requests to said memory), which NUMA node a process accesses the most, utilization percentage of each node's processing cores, data sharing between processes etc. An example of such a mechanism is that of [29].

A basic code-level infrastructure was created in our work, but no relevant experimental results will be presented, since the mechanism had enough design flaws that it could not be completed within the expected time frames. However, the code is available in the accompanying source code, to facilitate any possible attempt to extend it.

In figure 5.1, the described design is presented schematically.

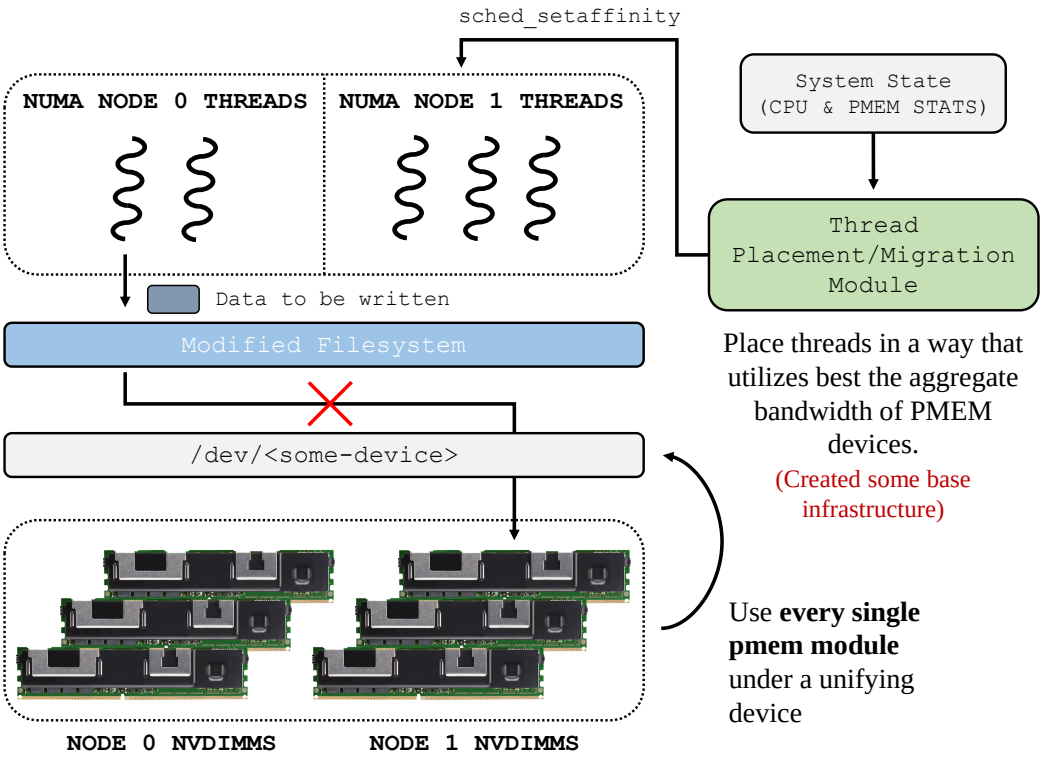

Figure 5.1: The infrastructure we aim to create

Chapter 6

Implementation

In this chapter we will present the development of the system proposed in chapter 5. We will see, in this order, how to create the single device that unifies all non-volatile devices of the system, the extensions made to ext4 for supporting NUMA aware allocations on the aforementioned device, the creation of the proper infrastructure in the kernel to collect I/O statistics per NUMA node, and finally, the basis created for the userspace component.

6.1 Integration of Non-Volatile Devices

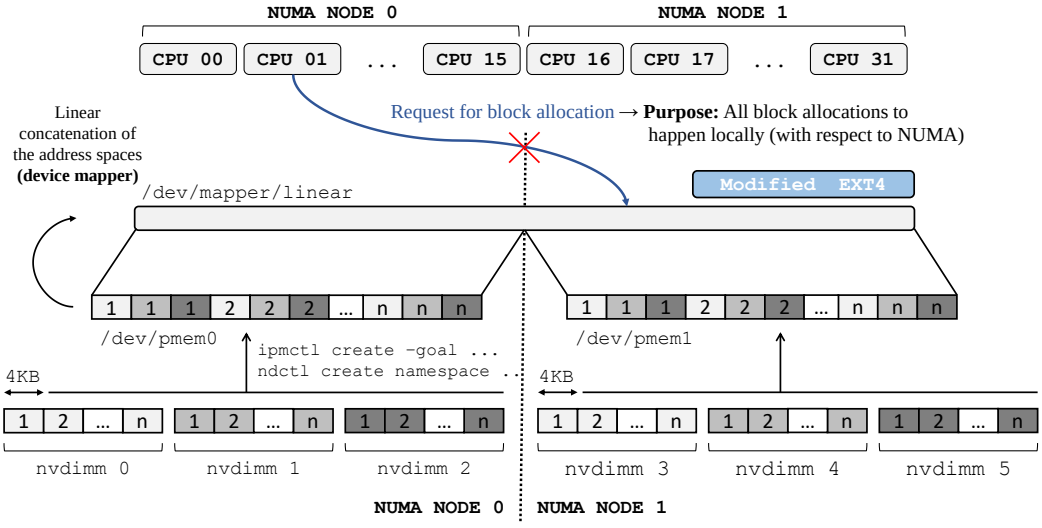

Figure 6.1: Schematic representation of how non-volatile devices are integrated into one

As we mentioned, we want for the purposes of this work to develop a sys-

tem that manages all non-volatile devices, regardless of their NUMA node. We recall that through Intel's IPMCTL tool we can obtain a hardware interleaving of the available devices per NUMA node. This is accomplished in a sufficiently efficient manner, so that there is no reason to manage devices at the level of a single NUMA node in any different way. So when we talk about unifying the devices hereafter, we mean combining the `/dev/pmem<X>` device files resulting from IPMCTL for each NUMA node.

In the direction of unifying the devices, we could provide multiple device files as parameters when creating the selected filesystem, and contain proper logic in its implementation to use the appropriate device for each access. An example of such an implementation can be seen in the extension of the NOVA file system performed for the purpose of comparison with the WineFS filesystem [18] [22]. However, this would add significant complexity to the modifications we are going to make at the filesystem level. Additionally, it would lead to a large logic overlap with extensively tested and readily available RAID drivers at the software level.

Based on what was mentioned, we reach the preference of creating a device file that takes care of device integration via some available Linux kernel driver. Then we create the file system on top of the new device, which will know how it was configured via the added modifications. The driver we choose to use is the device mapper, which is a framework for mapping multiple block devices into abstract, logical block devices [12]. We can do the mapping in a variety of ways, with the most well-known methods being the linear and striped mapping of devices (dm-linear and dm-stiped mapping respectively [10]).

Among the available methods of combining the devices, we choose the linear one, since it offers a clear separation between the input devices of the device mapper. This also greatly facilitates expressing in code how the devices have been unified, since it is sufficient to record the separation points of the various devices that make up the linear device. Obviously, with a striped mapping we could utilize to some extent the cumulative bandwidth of all the devices across NUMA nodes, since each access larger than the striped mapping's chunk would be served by more than one devices simultaneously. But this imposes the uniform utilization of the devices, thus a permanent burden on the interconnection network of the processors, and does not provide any easy-to-use means for dealing with the problematic patterns mentioned in chapter *Experimentation*. To demonstrate this, in chapter *Evaluation* we provide comparisons of our implementation with the performance of a relevant striped device.

Based on what we have mentioned so far, we get the overall picture presented in figure 6.1. It remains to specify some of the details on how exactly we unify the `/dev/pmem<X>` devices: As we will see later on, the search unit for allocations, at least at an initial stage, is the block group. Therefore, to be able to select

the destination node efficiently, it is sufficient to assign each block group to one of the individual nodes. However, in order for the mapping to be naturally unambiguous, we make sure that no block group is shared between addresses of different nodes. The direct way to achieve this is to make sure that the size we use from each device is a multiple of the size of a block group.

```
block_size_bytes=4096
blocks_per_group=32768
bytes_in_group=$(( block_size_bytes * blocks_per_group ))
sector_size_bytes=512

pmem0=/dev/pmem0
size_0=`blockdev --getsz $pmem0`
pmem0_bytes=$(( sector_size_bytes * size_0 ))
# Trim the size of each device to be perfectly divisible
# in terms of groups. The function round_down is defined
# in the script
size_0=$(( $(round_down pmem0_bytes bytes_in_group) /
    ↪ sector_size_bytes ))

# Do the same for pmem1 ...

# Make the linear mapping
echo "0 $size_0 linear $pmem0 0
$size_0 $size_1 linear $pmem1 0" | dmsetup create linear
```

Example of configuring the unifying device by rounding the size of the individual devices by the size of a block group

The process of creating the single device is shown in `experiments/scripts/mount/mount_create_linear.sh` of this work's source code. To avoid using hard-coded values, we create the device once, jus to get the `block_size_bytes` and `blocks_per_group` values via `dumpe2fs`. Then, we recreate it having appropriately rounded the sizes of the devices `/dev/pmem<X>`. The resulting file is represented by the device file `/dev/mapper/linear`.

It is reasonable to question whether it is a good practice not to use the space that is clipped from each device, since it seemingly leads to uneven use and reduction in the durability of some part of the medium. However, as common in modern storage media, non-volatile memories have a mechanism for wear-leveling via internal address translation [31], which we believe to negate the problem we described earlier.

6.2 Modifications to Ext4

As we saw in chapter *Experimentation*, the presence of remote accesses to non-volatile memories can significantly degrade their performance. Our first

and most important step towards circumventing this problem is to introduce the concept of NUMA locality to allocations performed by the filesystem of interest, in this case ext4. The purpose of this section is to present the relevant modifications made in this direction. The following paragraphs refer to some general design elements of ext4 on which we rely for the shaping of said modifications.

When creating and appending data to a new file we have two stages: nominally, we have the allocation of an inode for the file, and the later allocation of (multiple at times) data blocks when this is necessary. In both cases, the initial search unit for any allocation is the block group, rather than individual blocks. Searching at the block group level allows, on one hand, an easier assessment of the suitability of an area of the storage medium for the purpose of multi-block allocation, and on the other hand faster searching and updating of structures related to the availability of blocks. Therefore, for the purposes of the modifications we will describe, we first need to separate the block groups defined by ext4 to the individual NUMA nodes. Later, we will be referring to this partition to select block groups for the allocations corresponding to the requested NUMA node at each time.

Ext4 by design respects the need for locality of a file's metadata and data. This is perceptible if we consider the characteristics of simple magnetic disks, for which sequential reads not characterized by proximity on the medium are capable of leading to greater rotations of the magnetic disks, thus delaying the transfer of the requested information. This design makes sense even for more modern storage media, since many solid-state drives (SSDs), as well as non-volatile memories (PMEM) discussed in this document, have temporary buffers for grouping accesses [31]. Therefore, the spatial locality of (meta)data can lead to a better utilization of these structures.

Based on the advantages of spatial locality that we have just described, the modifications that we are making to ext4 will aim at preserving the property of locality both between data and metadata of the same file, but also between different files when this is necessary, mainly in case said files are under the same directory. The above principle can only be violated when there is appending to a file from a process running at a different NUMA node, since we recognize that the NUMA locality of write accesses is of utmost importance for performance.

It is noted that, for this chapter, our purpose is mainly to provide an overview of the design choices made during expanding upon the selected file system. This means that a desirable property of the text is mostly to justify the most pivotal design choices, and not necessarily to describe the implementation fully. This clarification aims to justify the non-uniformity of the degree of depth in describing the various points of the implementation. For example, while we give a fairly detailed description of the code added or changed in header files `numa.h` and `ext4.h`, the analysis of the changes made to the (meta)data allocation routines

will be more abstract.

6.2.1 Additions to file `ext4.h`

We first look at the modifications made to the basic header file of ext4's source code, namely `ext4.h`. The goals of said modifications are:

G1. To define new constants and structures that will contain the necessary information to describe the NUMA topology (number of nodes and distribution of block groups among them)

G2. Make it possible to adapt the optimizations of section *Existing Ext4 Optimizations* of chapter *Theoretical Background* to the needs of NUMA awareness.

Goal [G1] is partially achieved by encoding the number of NUMA nodes of the system via a preprocessor symbol (`#define EXT4_NUMA_NUM_NODES 2`). This practice introduces portability problems of the compiled kernel between systems with different numbers of nodes. However, we overlook this drawback, since the purpose of the work is primarily to build a proof of concept rather than a complete implementation. Additionally, the correct method for identifying the number of nodes within ext4's implementation presents significant complications in the coding process.

To complete the requirements of [G1], we create a new structure named `ext4_numa_info`. It contains the following information:

I) the total number of NUMA nodes: Although practically identical to `EXT4_NUMA_NUM_NODES`, having this bit of information stored in a variable makes the code more readable

II) the first block group of each NUMA node

III) the total number of block groups in each NUMA node

Regarding the information of (II) and (III), two important assumptions have been made: The first is that the concatenation of the non-volatile devices, at the construction of the linear device, is done respecting the kernel's ordering of the NUMA nodes. For example, the lowest addresses are occupied by the devices of NUMA node 0, the immediately lower addresses are occupied by the devices of node 1, and so on. This is easy to ascertain via the device mapper, and we adopt it as an assumption because it makes the implementation quite easier.

The second assumption is that it is not unlikely having a block group defined of the linear device being shared between devices of different NUMA nodes. As

we'll see next, we separate the block groups of the linear device between the NUMA nodes based on the validity of the first assumption and the fact that the devices /dev/pmem<X> we're concatenating are exactly the same size. Therefore, if we have for example an odd number of total block groups in a system with an even number of nodes, we see that the existence of a shared block group between NUMA nodes is certain.

Recognizing the validity of the second assumption, we are led to the need to formulate a rule for solving block group sharing issues, which will assign any shared block group to a single node. The simple rule chosen is to assign the shared block group to the node with the lowest identifying number. For example, if we have a block group that is shared between NUMA nodes 0 and 1, it will be assigned to node 0. The alternative solution that allows us to consider the block group as shared between the two nodes introduces significant implementation complications, and so we reject it. It is noted, however, that an attempt is made to eliminate shared block groups during the construction of the linear device, as seen in section *Integration of Non-Volatile Devices*.

Based on the above, we understand that we cannot ensure with absolute certainty that all NUMA nodes will have the same number of block groups. For this, we store the number of block groups per node in field (III), instead of simply assuming that all nodes have the same number of block groups, in which case we could simply use a single variable.

The information contained in struct `ext4_numa_info` is initialized via the new function `ext4_numa_super_init`. It's understandable that we'll be consulting the new struct quite often since we're aiming to make allocations that respect NUMA locality. It is therefore necessary to place the initialized structure in another, pre-existing ext4 structure that will make this information available at runtime as soon as possible. The appropriate structure to do the requested placement seems to be `ext4_sb_info`, which contains all the superblock information stored in memory ("data in memory") when ext4 is running.

We return to the preface of this subsection, in which we mentioned two goals for the modifications of file `ext4.h`. We move on to the analysis of goal [G2]. It is reminded that said goal is about adapting the pre-existing optimizations of ext4 to the needs of NUMA awareness. These optimizations, and the corresponding adjustments made, are as follows:

A) **Per inode list of preallocations**: Instead of having a single `i_prealloc_list` list per inode in the `ext4_inode_info` structure, we have an array of `EXT4_NUMA_NUM_NODES` of such lists, i.e. one for each node.

B) **Stream allocation:** As before, we now define per node the fields `s_mb_last_group` and `s_mb_last_start` of struct `ext4_sb_info`. These fields

contribute to the spatial continuity of the allocations made when doing stream allocation, and we expect this to be imposed separately on each node.

At this point, the coding convenience that comes from choosing to assign the number of NUMA nodes to a symbol of the preprocessor is made more clear. In any other case, we would need dynamic memory allocation for the aforementioned fields, which would add unwanted overhead and potentially lead to more sporadic memory accesses.

It is necessary to note for (A) that when writing the code it was assumed that assigning one `i_prealloc_lock` lock per node does not harm the correctness of the filesystem. This assumption was based on the fact that, theoretically, the corresponding lists cannot have any elements in common since they refer to allocations of different NUMA nodes. Additionally, we consider that there may be a race condition between multiple NUMA nodes each wanting to use their own preallocation for the same logical range of blocks of an inode. Such a condition is resolved early by locks held by the functions initiating the write accesses, which cover the entire inode. However, due to limited familiarity with issues regarding synchronization, this reasoning is likely to need revision.

Finally, we mention one more change that does not concern any optimization from those we saw in the chapter providing theoretical background, but a specific auxiliary parameter in the inode allocation process. This is about field `i_last_alloc_group` of structure `ext4_inode_info`, which also stops being in a single instance. This field helps in case the inode corresponds to a directory, so that we know in which block group metadata was last allocated for some child of said directory. During the next allocation, the recorded block group is examined first, thus favoring spatial locality. With our added modification, this property still applies, but for each `NUMA` node separately.

6.2.2 The new files `numa.h` and `numa.c`

In the newly introduced files `numa.h` and `numa.h` we attempt to hide much of the repetitive logic involved in restricting the block group search to a selected node. It is expected that a significant part of the work done upon the existing ext4 code is closely relevant to the general transformation presented afterwards. This transformation is important since it often appears in the existing ext4 codebase in many variants.

```
group = <some initial group>
ngroups = <total number of groups>
```

```
for (i = 0; i < ngroups; i++) {
  // work to be executed
  group = (group + step) % ngroups;
}
```

Before the transformation: Search all groups.

```
initial_node = <NUMA node where this is executed>

for_each_numa_node(iterator_node, initial_node,
    ↪ total_number_of_nodes) {
  // How many groups I have for this node
  n_node_groups = ext4_numa_num_groups(iterator_node);
  // Map this group to the current node
  group = ext4_numa_map_any_group(group, iterator_node);
  // For each group of this node
  for (i = 0; i < n_node_groups; i++) {
    // work to be executed
    group = ext4_numa_map_group(group + step, iterator_node);
  }
}
```

After the transformation: Search for groups by NUMA node, giving priority to the local one.

Obviously, some simplifications have been made for the purposes of the pseudocode given above, as there are other parameters in the code that deal with the implementation of the presented functions. By "mapping function" we mean a function that takes some input block group and uniquely maps it to some block group of the desired NUMA node given as input to the function.

We first state the fact that we have two mapping functions and not just one. The first, named `ext4_numa_map_any_group`, works for any given block group. The other function, named `ext4_numa_map_group`, is a faster and simplified version of the previous one, based on the assumption that the given block group has an identifier exclusively greater than or equal to the first group of the desired NUMA node. Distinguishing this case is particularly useful, since very often we only increment the current group during our search, and we are only interested in catching the case of overflowing into another node.

Functions like the ones shown, which belong to a broader class of new functions prefixed with `ext4_numa`, along with the symbol `for_each_numa_node`, are contained in the new file `numa.h`. Functions with limited size, i.e. all except `ext4_numa_super_init`, are defined in the aforementioned header file as `static inline`. We are aware that this is not necessarily a good programming practice, especially in terms of enabling easier code debugging. Nevertheless, its purpose is to reduce overhead from frequent function invocations that the C compiler probably couldn't mark as `inline` on its own.

Additionally, we want the NUMA awareness code to be enabled at the administrator's will when mounting the disk formatted with the modified version of the ext4 filesystem. We made sure this was set via the `-o numa` parameter. This flexibility will serve us greatly when comparing its original and modified versions, as we won't need to define a different filesystem containing the changes, or reboot with a different kernel each time for different subsets of the desired measurements. So having gathered as much as possible of the reusable logic for NUMA awareness in the functions of file `numa.h`, we manage to have more organized control as to whether we want to use our new code or not: it is enough to call `ext4_numa_enabled` at the beginning of the corresponding functions, and adjust the caller function's result accordingly.

To provide an example of the aforementioned design, instead of using symbol `EXT4_NUMA_NUM_NODES` directly at the code dealing with (meta)data allocations, we instead use function `ext4_numa_num_nodes`. It either returns the value of the above symbol if option `-o numa` has been given, or it returns 1 otherwise. That is, we consider by convention that we have only one node. This way, if we encounter later on the symbol `for_each_numa_node(..., num_nodes)`, and the `-o numa` parameter was not given during the append, it will do only a single repetition. So this actually negates the effect that the extra NUMA awareness code would have.

It should be noted that ideally, both from the point of view of completeness of work and as a proof of correctness for the design of the changes made, we should perform a formal proof for the ability of the modified filesystem to behave identically to its unmodified version in the absence of parameter `-o numa`. However, this is quite a complicated process and probably beyond the scope of this thesis, so we are content with the relative diligence that took place during writing the code, and the positive results observed during some makeshift testing procedures.

Going on, we provide an overview of the most important functions and symbols available through file `numa.h`. We assume that parameter `-o numa` is given when mounting, otherwise the values returned are consistent with the theoretical assumption that we have only one NUMA node.

- `for_each_numa_node(_n, var_node, init_node, num_nodes)`

  We are forced to include the helper variable `_n` in the symbol definition, although it is an implementation detail of the corresponding `for` iteration. Otherwise, if we declare the variable inside `for`, we get a compile-time warning. Also, the number of NUMA nodes is provided as a parameter precisely for the reasons we presented a few paragraphs ago.

- `bool ext4_numa_enabled(struct super_block *sb)`

Checks if parameter `-o numa` was given when mounting the disk containing the filesystem. It is used in almost all the following functions.

- `int ext4_numa_node_id(struct super_block *sb)`
  Returns the ID of the NUMA node on which the function is executed.

- `int ext4_numa_num_nodes(struct super_block *sb)`
  Returns the number of the system's NUMA nodes (equal to the value of symbol `EXT4_NUMA_NUM_NODES`).

- `int ext4_numa_bg_node(struct super_block *sb,`
  `↪ ext4_group_t g)`
  Returns the NUMA node to which the given block group corresponds.

- `ext4_group_t ext4_numa_map_group(struct super_block *sb,`
  `↪ ext4_group_t group, int map_node, ext4_group_t ngroups)`
  Returns the mapping of the input group `group` to the input node `map_node`. We assume that the given group necessarily belongs either to the requested node, or to someone with a larger ID.

- `ext4_group_t ext4_numa_map_any_group(`
  `↪ struct super_block *sb, ext4_group_t group,`
  `↪ int map_node)`
  Like the previous function, but without the mentioned limitation.

- `ext4_fsblk_t ext4_numa_map_any_block(`
  `↪ struct super_block *sb, ext4_fsblk_t block,`
  `↪ int map_node)`
  Like `ext4_numa_map_any_group`, except that the mapping is done at block level instead of the block group level. This is useful in ensuring the robustness of certain changes made to the filesystem. Specifically, it aids in the case that we have been given a block as a target (as happens in an allocation request) which due to a previous failure does not respect the property of NUMA locality. With the help of this function, we make sure that it will ultimately be mapped to the desired node.

- `int ext4_numa_block_node(struct super_block *sb,`
  `↪ ext4_fsblk_t blk)`
  Returns the NUMA node of the given block.

- `void ext4_numa_super_init(struct ext4_sb_info *sbi)`
  Called inside `ext4_fill_super` (in `super.c`) to initialize the information

of structure `ext4_numa_info` contained in `ext4_sb_info`. Some policies regarding the process of filling the relevant fields have already been previously described, and the fine-grained details are hopefully adequately covered in the given implementation by the relevant comments accompanying this function.

With this reference, the reader is hopefully in a better position to read and understand more easily the logic and design of the changes made to ext4, as those will be described below. In the next two subsections we will see as abstractly as possible the modifications and additions made to introduce NUMA awareness features, first in inode allocation and then in multiblock allocation.

6.2.3 Inode Allocation

By utilizing the methods presented in section *Tracing in the kernel*, we trace the sequence of functions for metadata allocation in case of creating a new file.

```
do_sys_open        → do_sys_openat2    →
do_filp_open       → path_openat       →
open_last_lookups  → lookup_open       →
ext4_create (ptr)  → __ext4_new_inode
```

The `(ptr)` notation above specifies that the corresponding function was found via some pointer. These cases are mentioned because it is not always easy to determine where such a function is called from, at least by searching with tools like Elixir Bootlin. Going over the presented sequence of functions, we focus on `__ext4_new_inode`, as it looks the most promising.

In figure 6.2 we have an abstract, schematic presentation of inode allocation in the original version of ext4. We see that there is a difference in the behavior of the inode allocation function depending on whether we have a request for an inode of a directory or a file.

It is important to mention that during the following modifications no priority was given to maintaining support for the flex group feature (which we saw in section *Existing Ext4 Optimizations* of chapter *Theoretical Background*). For this reason, parameter `-o numa` must be accompanied by parameter `-o ^flex_bg` when mounting a disk with the modified filesystem, which is checked in `ext4_fill_super`. While this does hurt the completeness of our modifications, the intent was to better allocate time so as to preferably get the mutliblock allocation changes right. However, the adjustments required to fix this deficiency are not overly complicated, and could be part of a future extension of this thesis' work.

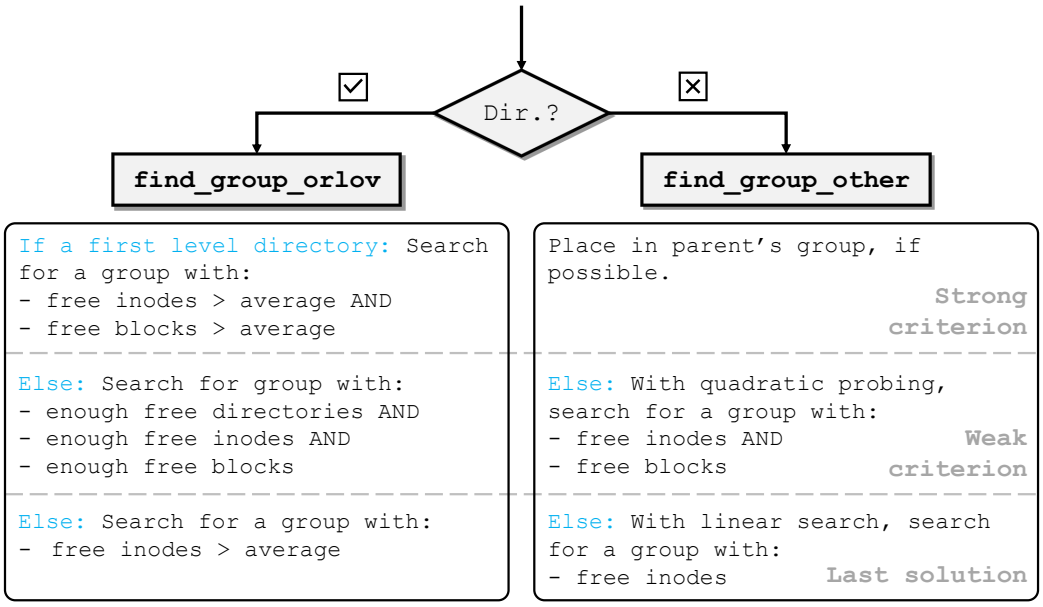

Figure 6.2: Abstract representation of function `__ext4_new_inode` in its original version

In the case of a directory, the so-called Orlov algorithm is used. Significant simplifications have been made in our schematic representation, and any logic including the optimizations of flex groups is not presented at all here since it is not supported by the modifications we have made. For further study of the behavior of the algorithm, we refer to the comments accompanying the function in the source code of kernel 5.13, which are quite informative as they fully outline this routine. Also, as with the rest of the functions we'll be looking at, the changes made for the purposes of our work can be seen directly via the following command:

```
$ git diff 32debcbccb4c93a3f3b188e9e8eb15d5e9951e37 HEAD
    ↪ fs/ext4/ialloc.c
```

String `32debcbccb4c93a3f3b188e9e8eb15d5e9951e37` corresponds to the hash of the first commit, found via command `git log`, and `HEAD` corresponds to the last commit. We have taken proper care to accompany these changes with the necessary comments for their being more easily understood.

In summary, the goal of the algorithm is to first fairly distribute among the groups the entries in the first level of the directory tree, by examining how many free inodes and blocks there are. This is to ensure uniform utilization of the storage medium. In case the search failed or no first level directory was given,

we do a linear search of the groups to find one that satisfies a series of strict criteria, including the availability of several inodes, directories and blocks. If the previous search turns out to be unsuccessful, we relax the search criteria, proceeding to find the group with relatively more free inodes. So we have two phases of the algorithm that follow a strict set of criteria, and a third phase that executes if the previous two fail.

The most natural adaptation of Orlov's algorithm, in order to add NUMA awareness features, is as follows: On one hand, we limit the search of the first two phases to the groups of the local NUMA node (i.e. the one from which the function is called). On the other hand, we leave the search done during the third phase free to expand to other nodes, but only in case we find that there is no suitable group available in the current node. We will see right away that the changes made in the case of file metadata allocation have a similar structure.

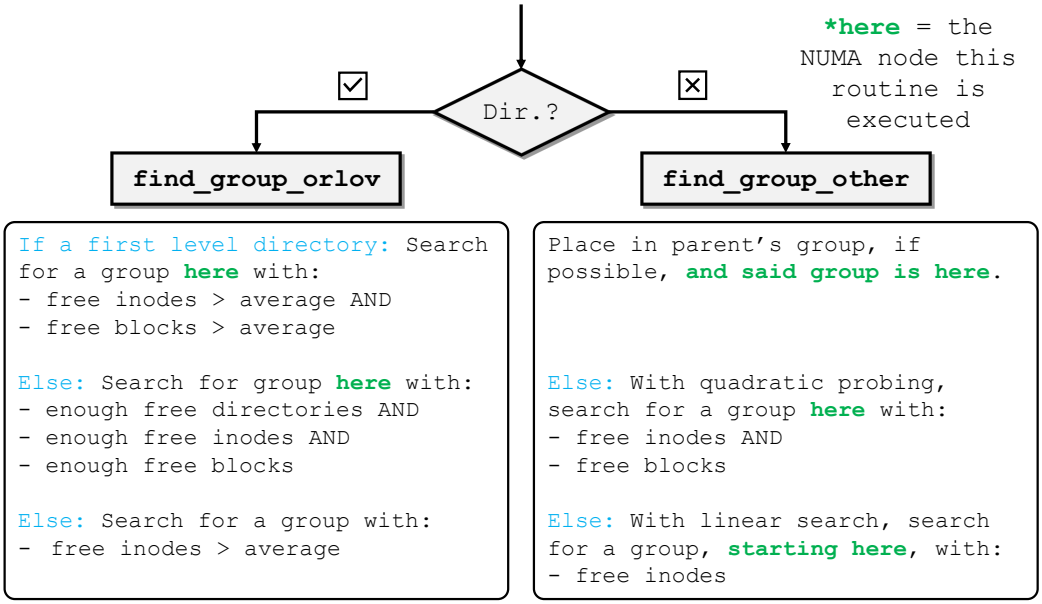

Figure 6.3: Abstract representation of function `__ext4_new_inode` after our changes

For the `find_group_other` function that takes care of finding the group to allocate a new file's inode, we follow some slightly different logic. The first factor we examine is whether the new inode can be successfully placed in the group of its parent, i.e. the directory to which the corresponding file belongs. In case this is not possible, we are taken to the second stage of the algorithm, in which we use quadratic probing given the group g of the parent, to find a group that has both free inodes and free blocks. By the term quadratic probing we mean that in iteration i of our search we check group $(g + i^2)$ (in arithmetic modulo the

number of groups). Its purpose is to potentially separate files of non-identical directories that happen to share the same group. If the previous search fails, we proceed to the third and final stage, in which we linearly search the groups with the looser criterion of the simply having available free inodes.

As in Orlov's algorithm, we restrict here as well the first two phases to the current node, while allowing the third and last phase to expand the search to include other nodes if no group of the current one is deemed suitable. It is noted that the behavior of the algorithm is different if we have the flex group optimization on, in which case we call Orlov's algorithm with suitable parameters upon failing to place the new inode in its parent's group. Therefore, to ensure compatibility between the added NUMA awareness features and the optimization of flex groups, special care is needed in the case we just described.

6.2.4 Multiblock Allocation

Using the methods of section *Tracing in the kernel*, we obtain the following sequence of calls for when writing to a file resulting in data blocks being allocated:

```
vfs_write                → new_sync_write        →
call_write_iter          → ext4_file_write_iter  →
ext4_dax_write_iter      → dax_iomap_rw          →
iomap_apply              → ext4_iomap_begin      →
ext4_iomap_alloc         → ext4_map_blocks       →
ext4_ext_map_blocks      → ext4_mb_new_blocks    →
ext4_mb_regular_allocator
```

After going through the code of the relevant functions, we recognize that the last three are more interesting for our purposes of adding NUMA awareness features:

- `ext4_ext_map_blocks`
  Search in the extent tree to determine if we can use an existing extent of the file (for overwriting) or a reserved cluster that overlaps with the ones close to the requested logical extent of the file (for appending). If these checks fail, we call `ext4_mb_new_blocks` to find a suitable usable block.

- `ext4_mb_new_blocks`
  We first check the preallocation space that has resulted from the optimizations we went over at the chapter providing theoretical background. If

there are no suitable preallocations, we make entirely new allocations using the following function.

- `ext4_mb_regular_allocator`
  This function looks for an appropriate block group, i.e. the one that will offer the most ideal available sequence of blocks for the purposes of the requested allocation.

Note that we have two possible design choices for how to select a NUMA node when appending data to a file: the first one always selects the node on which the writing thread is executed, while the second one prefers the node on which the inode of the file we are appending to is located. One design favors the NUMA locality of accesses, while the second favors the locality of the file's data blocks. Based on the results of the experiments presented in previous chapters, we choose to implement the first design. However, a smarter implementation would probably incorporate the latter on occasion.

We study each function individually to determine how we can adapt them for the purposes of our work.

Function ext4_ext_map_blocks

It is clarified that with the terms "logical block" and correspondingly "logical (block) range" of a file, we refer to the blocks that it either already occupies or is about to allocate, numbered in the order they appear in the file. For example, logical block 0 of a file corresponds to the data at its beginning, and can correspond to any block in the file system, regardless of its logical number. Logical block 1 corresponds to the immediately following data of the file, etc.

We start with the function that maps the requested range of logical blocks of the file to the extent tree (`ext4_ext_map_blocks`). Everything we will describe is presented in the schema of figure 6.4. First, we try to see if there is already an extent that covers the requested logical blocks, which case corresponds to a call for overwrite. Since we are interested in the case of allocating new blocks, we proceed to the next check.

At this point we have not found an extent that covers the entire requested logical block range, but we have located the closest leaf of the extent tree, i.e. the one corresponding to the immediately previous reserved logical range. This leaf (actually the corresponding extent) probably holds some cluster that is not fully utilized. As we said in section *Existing Ext4 Optimizations* of the chapter going over the theoretical foundation, because of cluster allocation the smallest allocation unit is not an individual block, but groups of blocks of some selected fixed size, the so-called clusters. This way, at some point we may reserve more blocks than we need. We are therefore examining whether this is exactly our

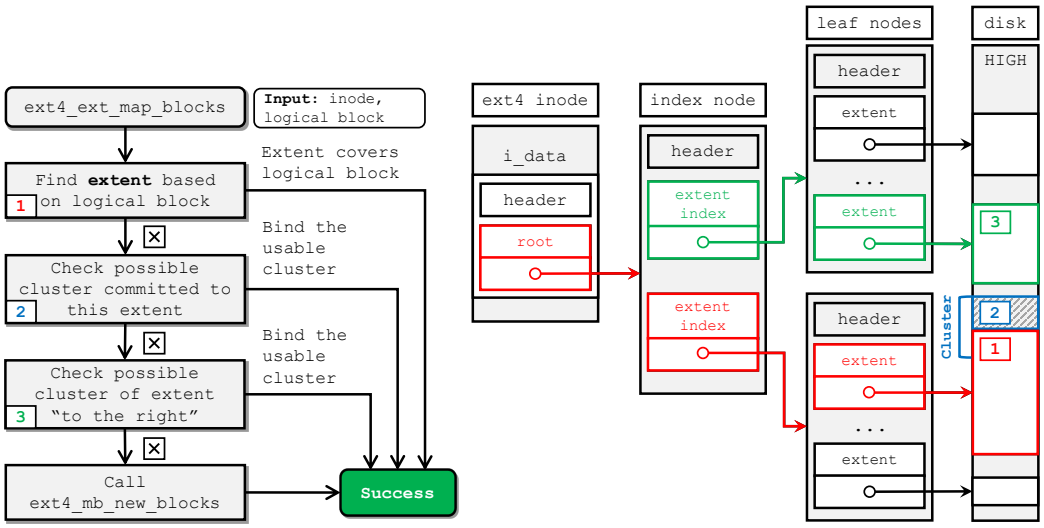

Figure 6.4: How function `ext4_ext_map_blocks` works

Note: In case (3) as seen in the figure, the extent itself is colored, but it is understood that we are simply checking its partially utilized clusters, as in step (2).

case, so that we can satisfy part of the request through the available blocks of the cluster. If this is not the case, we do the same check for the immediately following extent (the one to the "right") of the current one.

If all the previous checks have failed, we come to the realization that we need to use available blocks that do not correspond to the extent tree, which we attempt with `ext4_mb_new_blocks`.

We point out that steps (1), (2) and (3) of figure 6.4, which are concerned with examining the extent tree at the time of calling `ext4_ext_map_blocks`, may open up the risk of remote block allocation. The reason is that we may first bind a cluster of one NUMA node partially, and later on utilize it further during a block allocation request from a completely different node for the immediately next logical blocks. It is crucial that we fill the clusters under any circumstances, because otherwise the optimization will start yielding unusable blocks in the reserved clusters. So, we recognize that there is nothing we can do about this case to prevent remote block allocations, other than perhaps turning off cluster allocation altogether. However, we believe that this will have a significant impact on filesystem performance, and therefore reject it completely as an idea.

We conclude with the preceding reasoning that we cannot change much in the way function `ext4_ext_map_blocks` works, so we focus on how we can adapt `ext4_mb_new_blocks` instead to respect NUMA locality.

Function ext4_mb_new_blocks

Figure 6.5 shows a simplified schematic flowchart representation of `ext4_mb_new_blocks` before the relevant modifications.

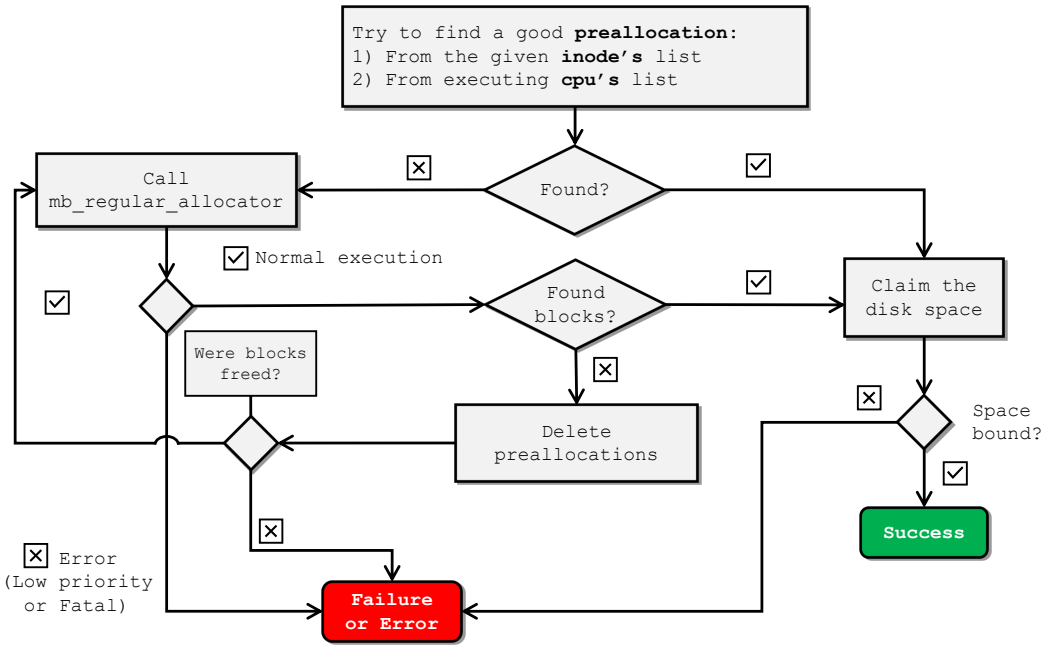

Figure 6.5: Flowchart of `ext4_mb_new_blocks` in its original version

The first step is to check if there is any usable preallocation available, by calling `ext4_mb_use_preallocated`. We initially check `i_prealloc_list` of the specific inode. If it has any preallocation that fully covers the allocation request's range of logical blocks (expressed through parameter `ext4_allocation_context` of `ext4_mb_use_preallocated`), then we use it.

If there is no suitable preallocation for the specific inode, and the request is small enough (in terms of the size of the required space) to have a hint for group allocation (flag `EXT4_MB_HINT_GROUP_ALLOC`), then we examine the locality group of the specific processing core on which the function is executed. If there are usable preallocations, in the sense that the length of the request can be covered, we choose the one that is closest to the target block of the given request.

If we find that there is no suitable preallocation, we proceed to allocate new clusters by calling function `ext4_mb_regular_allocator`. In the event that the aforementioned function did not present an error but at the same time did not satisfy the given request, we try to delete as many preallocations as needed to free up the space required by the request. In case this succeeds (function `ext4_mb_discard_preallocations_should_retry` is evaluated to be true), we try

calling ext4_mb_regular_allocator again.

Figure 6.6 shows the same flowchart, this time including the relevant modifications. A first obvious change is the restriction of the preallocation search to the NUMA node at which the function is executed. In theory, this sounds simple enough to implement. In practice, however, special attention needs to be paid to the separation of preallocations between nodes, because the ext4 codebase has been structured in a way that makes it quite robust in terms of identifying cases in which its logic is violated.

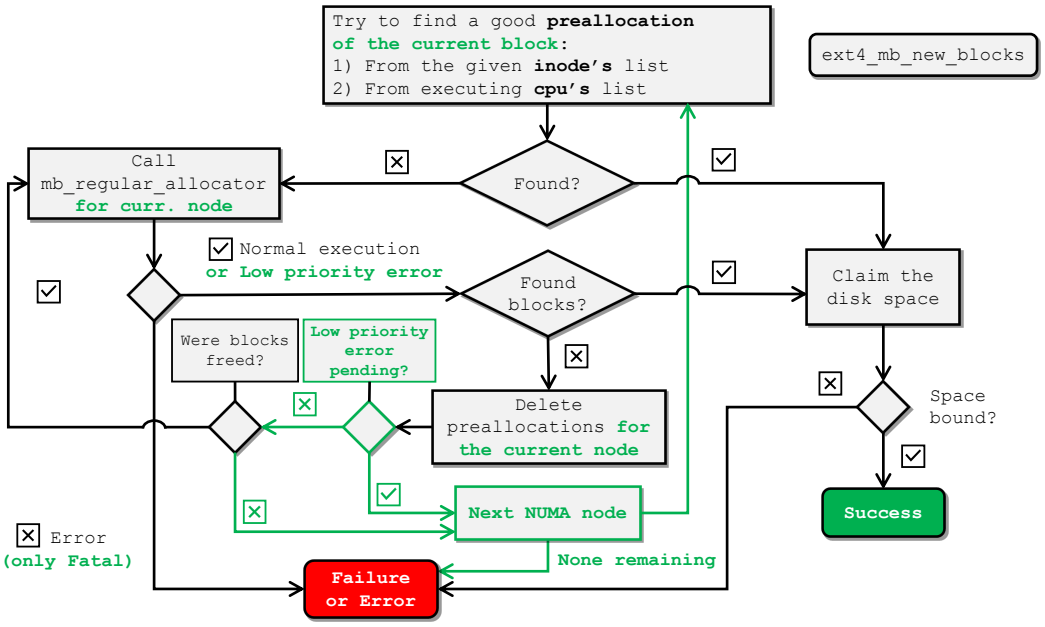

Figure 6.6: Flowchart of function ext4_mb_new_blocks after our modifications

For example, if we choose to ignore any available per-inode preallocation during ext4_mb_use_preallocated, simply because it does not correspond to the desired NUMA node, then said function will be unsuccessful with high probability. Afterwards we'll prepare for ext4_mb_regular_allocator by first calling ext4_mb_normalize_request. In the latter function, however, a BUG_ON macro checks that there is no preallocation for our specific inode that already covers the requested range of logical blocks. Therefore, ignoring some valid per-inode preallocation in ext4_mb_use_preallocated will almost certainly result in an error later during the execution.

To avoid this problem, we make sure that we have separate lists for per-inode preallocations, one for each NUMA node, and that the desired node is a parameter of the functions mentioned in the immediately preceding paragraph.

For locality group preallocations, the described issue does not occur, as we

can simply ignore the relevant preallocations of other nodes. Of course, this practice can theoretically lead to unnecessarily examining preallocations that concern other nodes and which we cannot use. However, we recall that the list of locality group preallocations is per processing core. Therefore, if we take care that the allocations via `ext4_mb_regular_allocator` respect the property of NUMA locality, then no preallocation involving non-local NUMA nodes can occur. Even if an allocation is made to a non-local NUMA node, due to an error in how our modifications work, the possibly generated preallocation has the prospect of being deleted if at some point there is not enough space for a future allocation. Therefore, no preallocation can be stale forever, thus it cannot prevent us from fully utilizing our available space.

The most complex part of the changes made to `ext4_mb_new_blocks` was keeping the error-handling logic of `ext4_mb_regular_allocator` intact. As seen in the following flowchart, `ext4_mb_regular_allocator` has two types of errors:

- **Low-priority errors:** They can occur by calling `ext4_mb_good_group_nolock` for a particular group, but these errors are not critical enough to prevent us from continuing to search for another candidate group. If a group is found at the end of `ext4_mb_regular_allocator`, the low-priority error is ignored, otherwise it is returned to the calling function.
- **High priority bugs:** Related to a malfunction in the buddy allocator. If such an error occurs, we stop the search immediately and return the error to the calling function.

For the `ext4_mb_new_blocks` function in its original form, however, there is no differentiation between the aforementioned errors, and if it receives an error from `ext4_mb_regular_allocator`, it reports it immediately regardless. We, on the other hand, change the order of some functions of `ext4_mb_new_blocks`, as we have the transformation shown in figure 6.7. The extra code due to the transformation is marked in blue. We note that many important simplifications have been made for the operation of `ext4_mb_new_blocks` in the given pseudocode. Just the absolutely necessary information is provided to show the need for more detailed error handling within the examined function.

So now we can't just treat low and high priority errors in the same way, and we need to introduce the logic of this differentiation in `ext4_mb_new_blocks`, despite said logic being originally only contained in `ext4_mb_regular_allocator`. We understand that we can temporarily ignore a low-priority error until the search for all NUMA nodes is complete. So we report it only after the search has failed on all nodes.

ext4_mb_new_blocks: Pseudocode of original version

```
SEARCH_PREALLOCATED(request)
if request → status ≠ FOUND then
    try :
        REGULAR_ALLOCATOR(request)
    catch Error :
        return Error
if request → status = FOUND then
    return request → newblock
else
    return −ENOSPC
```

ext4_mb_new_blocks: The changes after the transformation

```
for node n ∈ NUMA nodes do
    SEARCH_PREALLOCATED(request)
    if request → status ≠ FOUND then
        try :
            REGULAR_ALLOCATOR(request)
        catch Error :
            if Error is critical then
                return Error
            else
                record Error
    if request → status = FOUND then
        return request → newblock
if Error recorded then
    return Error
else
    return −ENOSPC
```

Figure 6.7: The transformation for `ext4_mb_new_blocks`

Special care is needed not to violate a certain property of `ext4_mb_new_blocks`, which is not apparent in the pseudocode due to the simplifications made. Specifically, in its original version we are allowed to retry from the point of calling `ext4_mb_regular_allocator` if enough preallocations can be discarded for this to be meaningful (`ext4_mb_discard_preallocations_should_retry`). However, in the original code we could not have reached this check if a low-

priority error had preceded it, as it would have been reported immediately. So, in the modified version of `ext4_mb_new_blocks` we stop doing this check once a low-priority error is recorded for any node. Otherwise it is assumed that undefined behavior may result from double searching in the same node while the error is pending.

In the source code accompanying this work, comments have been provided that further explain the individual vagaries of the changes made to `ext4_mb_new_blocks`.

Function ext4_mb_regular_allocator

Having already mentioned several details of its implementation, we also present the changes made to `ext4_mb_regular_allocator`, and by doing so we complete this section describing the changes made to the multiblock allocation logic of ext4. Table 6.1 also analyzes the criteria mentioned in the figure.

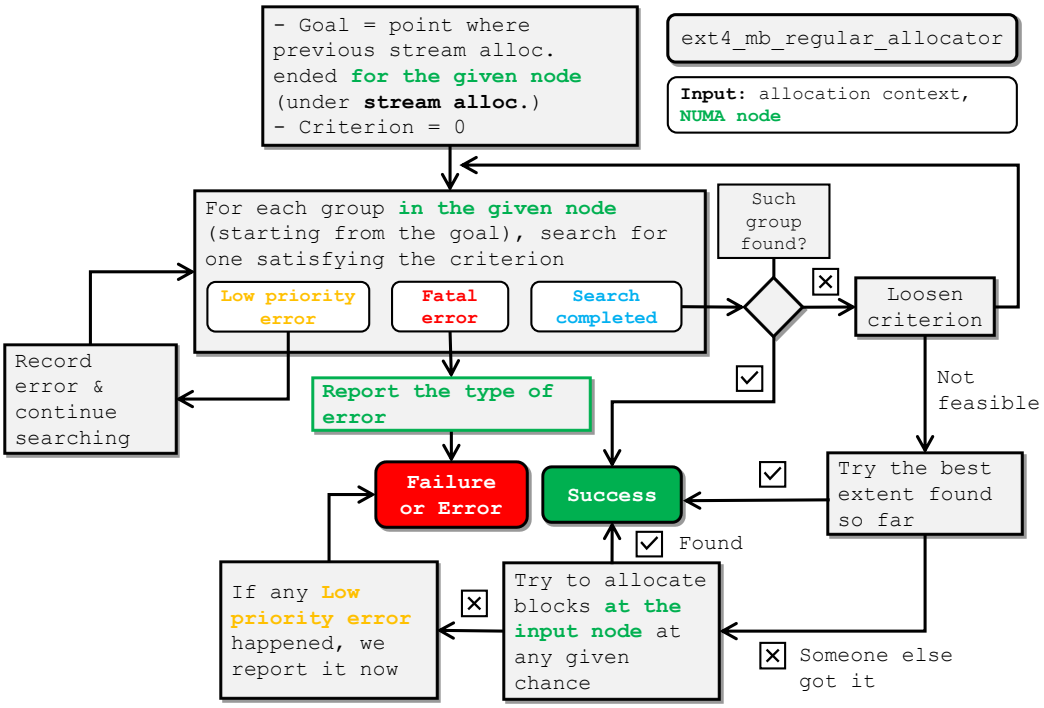

Figure 6.8: Flowchart of `ext4_mb_regular_allocator`
(modifications are shown in green)

Although the flowchart looks complicated, this is actually due to the inherent complexity of the function. The modifications we need to make are quite simple:

Criterion	Condition for the group (or request)
0	As in criterion 1, but request length is a power of 2
1	There is a fragment (range of contiguous blocks) with length ≥ Requested length
2	Total available blocks ≥ Requested length
3	No condition

Table 6.1: Explanation of the criteria used by `ext4_mb_regular_allocator`

- If we are doing stream allocation for the current request, we choose the target corresponding to the NUMA node given as a parameter.
- We limit the scope of our search to the NUMA node given as a parameter.
- If a high-priority error occurs, we notify the calling function via the newly introduced `fatal_err` parameter.

6.3 NUMA IO Accounting

We ideally want the kernel to provide an interface to users through which per-process and per-NUMA node Input/Output (I/O) statistics are available for individual processes. This can primarily serve the userspace component, since a key part of its decision-making logic is to find the node to which each process makes the most accesses. But more than that, this interface can be a handy tool for debugging when extending ext4: it is noted that many corrections were made by periodically observing the percentage of accesses by filebench threads to unwanted NUMA nodes, with the aid of this interface.

This interface is very similar to file `io` of special-purpose filesystem proc. The problems with this file are 1) that it does not record I/O statistics for the DAX operation, and 2) that the logging it does is not at the level of NUMA nodes, i.e. the information it provides is cumulative for all nodes. Therefore, building on the infrastructure of the procfs `io` file, we aim to add a `numa_io` file that will fix the deficiencies mentioned. The reason why we create a new file and do not expand the existing one is mainly related to maintaining compatibility with installed system programs whose correct operation may depend on the specific structure of the `io` file.

Immediately after we examine the basic steps of the implementation of this new file.

6.3.1 Registering the new procfs file

Our first step is to check the implementation of the `io` file in procfs' source code. This way, we manage on one hand to determine from which structure of the corresponding process the relevant statistics are drawn, in order to expand it appropriately. On the other hand, we can be informed about the methodology of inserting new files into procfs.

After some searching, we are taken to kernel's `fs/proc/base.c` file. In its code there are two structures with global scope:

- `struct pid_entry tgid_base_stuff`: This contains entries for the procfs files under directory `/proc/<pid>/task/<tid>/`
- `struct pid_entry tgid_base_stuff`: This structure contains the corresponding entries of directory `/proc/<pid>/`

In the example of file `io`, its implementation for each directory differs, as shown by the following lines in `fs/proc/base.c`:

```
// static const struct pid_entry tgid_base_stuff[] = { ...
ONE("io", S_IRUSR, proc_tgid_io_accounting),
// static const struct pid_entry tid_base_stuff[] = { ...
ONE("io", S_IRUSR, proc_tid_io_accounting),
```

For better understanding the semantics of these functions, we clarify that by convention we consider each `<pid>` process to be associated with its `<tid>` child threads. Among them is the main thread of the process, which takes `<tid>` equal to `<pid>`. In reality, however, it is parenthetically mentioned that in the kernel's implementation there is only the concept of process, which has a unique PID (Process ID) and a TGID (Thread Group ID). The entity we considered as a process with id `<pid>` has TGID = PID, while the child threads have each their own PID, but share the TGID with their parent.

In the case of file `/proc/<pid>/task/<tid>/io`, the I/O statistics of only one thread with ID `<tid>` and parent `<pid>` are available (`<pid>` = `<tid>` may also apply). In the case of `/proc/<pid>/io`, we have cumulative I/O statistics for all threads under the parent `<pid>` (as we have already mentioned, this includes the main thread of the process that takes `<tid>` = `<pid>`). For example, if we wanted to get the statistics of the process itself and not its children, we would look at file `/proc/<pid>/task/<pid>/`.

From the above, it is immediately clear how we can add the new files `/proc/<pid>/task/<tid>/numa_io` and `/proc/<pid>/numa_io`, and what we need to pay attention to in their semantics, at least for them to be consistent with the logic of file `io`.

6.3.2 Introducing NUMA I/O Accounting Fields

Looking at the implementation of functions `proc_tgid_io_accounting` and `proc_tid_io_accounting`, we soon find out struct `task_io_accounting` via fields `read_bytes` and `write_bytes`. We are taken to file `/include/linux/task_io_accounting.h` of the kernel's source code, to which we make the following additions:

```
// struct task_io_accounting { ...
u64 numa_read_bytes[2];
u64 numa_write_bytes[2];
// ...
```

For each type of access (read or write) we have a field that is an array of values, with each value corresponding to a NUMA node. Ideally, we would like a single parameter between the ext4 implementation and the files in directory `/include/linux/`, which would represent the number of NUMA nodes in the system (role similar to that of constant `EXT4_NUMA_NUM_NODES`, but less narrow). At the time of writing this document, it was not immediately apparent how this is possible in an elegant way or in accordance with the infrastructure provided by the kernel's compilation system. For example, it would be preferable for us to set a related parameter can be configured via command `make menuconfig`. However, the time limits in developing our codebase did not allow the investigation of this eventuality, so it was chosen in this case for the number of NUMA nodes of the system to remain a hard-coded value.

6.3.3 Functions to update the new fields

Our next step is to determine the place within the kernel that fields `read_bytes` and `write_bytes` of struct `task_io_accounting` would be updated. After searching, we find functions `task_io_account_read`, `task_io_account_write` and `task_io_accounting_add`. For better readability of our implementation, we add the direct analogues of these functions, that is, the functions of our source code prefixed with `task_numa_io_account_`.

We understand that function calls prefixed with `task_io_account_` indicate where we potentially need to add our new functions prefixed with `task_numa_io_account_`. After looking through with `grep`, and ignoring cases of filesystems that do their own I/O accounting directly, we end up with the functions of the following table.

From these subcategories, for our implementation we will mainly focus on "Memory Management" and "Block Device," since "Direct-IO" is not of particular interest for our purposes. However, how it can also be included will hopefully

Category	File	Function
Memory Management	/mm/readahead.c	read_cache_pages
	/mm/page-writeback.c	account_page_dirtied
Block Device	/block/blk-core.c	submit_bio
	/fs/block_dev.c	__blkdev_direct_IO
		__blkdev_direct_IO_simple
Direct-IO	/fs/direct-io.c	submit_page_section
	/fs/iomap/direct-io.c	iomap_dio_bio_actor

Table 6.2: Points in the Linux kernel where I/O accounting takes place

become apparent by the end of this section. We notice that no category seems to reference the DAX feature. As mentioned, I/O accounting is not supported at all for DAX in the original version of kernel version 5.13. So, we first need to locate the appropriate entry point for the new IO accounting functions. Looking at the sequence of kernel functions associated with DAX accesses, we find the candidate function `dax_iomap_actor` of file `fs/dax.c`.

However, functions with the prefix `task_numa_io_account_` need as a parameter the identifier of the NUMA node whose statistics we want to update. Therefore, in any function we are interested in logging I/O statistics, we must have a way to retrieve the correct NUMA node based on its parameters. This procedure turns out to be non-trivial, and so we present the individual subcases below.

NUMA node retrieval in code regarding DAX operations

Function `dax_iomap_actor`, in which we want to record I/O statistics, is given as a parameter an object of type `dax_device`, and inside its iteration the relevant `sector` of the device is calculated. So we want to create a function called `get_numa_node` that will combine these two pieces of information to give us the corresponding NUMA node.

Since function `get_numa_node` depends on the driver corresponding to the object of type `dax_device`, it makes sense to define it as a member of the abstract struct `dax_operations`. The description of this structure can be found in `include/linux/dax.h`. Generally, and intuitively, we will see that many abstract interfaces tend to be in directory `include/linux`. This way, we have constructed the interface that will be called inside `dax.c`, and which the device mapper driver will implement accordingly.

Our next step is the implementation of `get_numa_node` in the driver we are

interested in. Given a request to a device mapper logical device, the driver's general logic of operation is to initially determine the target device (object of type `dm_target`) corresponding to the given offset of the mapped device. It then satisfies the request on the chosen target, in a way that depends on the type of mapped device (linear, striped, etc). To better demonstrate the described logic, the following is the example of function `dm_dax_copy_from_iter` of file `drivers/md/dm.c` in Linux kenrel's source code.

```
static size_t dm_dax_copy_from_iter(struct dax_device
    ↪ *dax_dev, pgoff_t pgoff, void *addr, size_t bytes,
    ↪ struct iov_iter *i)
{
        struct mapped_device *md = dax_get_private(dax_dev);
        sector_t sector = pgoff * PAGE_SECTORS;
        struct dm_target *ti;
        long ret = 0;
        int srcu_idx;

        ti = dm_dax_get_live_target(md, sector, &srcu_idx);

        if (!ti)
                goto out;
        if (!ti->type->dax_copy_from_iter) {
                ret = copy_from_iter(addr, bytes, i);
                goto out;
        }
        ret = ti->type->dax_copy_from_iter(ti, pgoff, addr,
            ↪ bytes, i);
 out:
        dm_put_live_table(md, srcu_idx);

        return ret;
}
```

The job of determining the target is handled by calls to `dm_dax_get_live_target` and `dm_put_live_table`. Among them, satisfying the request on the specific target type is done through the `type` field (with type `target_type`) of object `dm_target`. We understand that we must first define some function `get_numa_node` at the abstraction level offered by struct `target_type` described in file `include/linux/device-mapper.h`. Then, we implement this function for the type of device we are interested in, usually the linear device which has its implementation logic in file `drivers/md/dm-linear.c`.

Having done what's mentioned above, the implementation of the desired function `dm_dax_get_numa_node` in `drivers/md/dm.c` is completely analogous to the example function `dm_dax_copy_from_iter`. We assign its pointer to the `dm_dax_ops` object, and it can now be called as intended from `fs/dax.c`.

Note that the implementation of interface `get_numa_node` for objects

of type block_device is almost identical. It is enough to define this interface in the description of struct block_device_operations of file include/linux/blkdev.h, and implement it in drivers/md/dm.c. The implementation has so much in common with that of dax_operations, that we have defined a more general function dm_general_get_numa_node which is called by dm_blk_get_numa_node and dm_dax_get_numa_node.

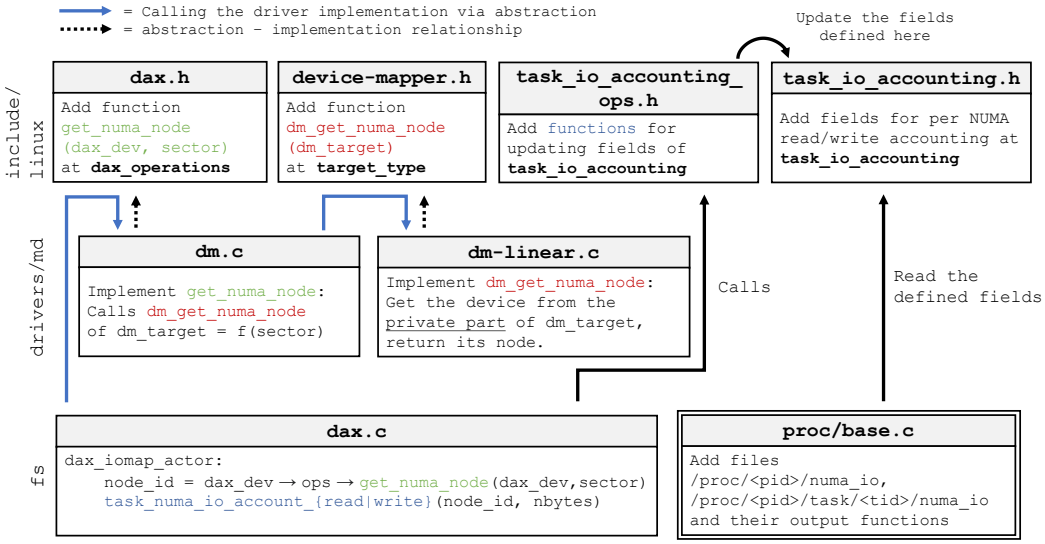

Figure 6.9: Implementation of recording I/O statistics per NUMA node, in a way that supports the DAX feature

Figure 6.9 shows everything we have mentioned so far, but not including the extension to support the interface get_numa_node for objects of type block_device.

Retrieving the NUMA node in code of the "Block device" category

In functions submit_bio, __blkdev_direct_IO_simple and __blkdev_direct_IO, either an object of type bio is given as a parameter, or it is constructed within it for the purposes of the function. Structure bio makes the sector to be accessed immediately available, so it is sufficient to simply use the get_numa_node interface, as provided for structure block_device. We described how to create the interface just before.

Retrieving the NUMA node in code regarding Memory Management

Both functions read_cache_pages and account_page_dirtied are given as a parameter a mapping of type address_space, and we work with pages

(objects of type `struct page`). From the given mapping, we can retrieve the relevant inode. If we specify a method to retrieve the sector from the given inode and page, we can later leverage interface `get_numa_node` of struct `block_device` to achieve finding the NUMA node.

Note that the mapping logic (`inode`, `page`) → `sector` is contained in the filesystem we use. So, we make a `page_sector` interface in the kernel, which takes care of doing the mapping, and can be implemented separately on each filesystem.

A good place to define the interface is struct `inode_operations` of file `include/linux/fs.h`. We implement the interface in ext4 via function `ext4_page_sector`, defined in file `fs/ext4/inode.c`, with logic based partly on the preexisting `ext4_mpage_readpages` function . Finally, we assign a pointer to the implemented function, to the new `page_sector` field of struct `ext4_file_inode_operations` of file `fs/ext4/file.c`.

With all the work described in this section, we now have the statistics we intended to make available through our new `numa_io` procfs files. Full support for Direct IO is missing, but that's something that's easily implemented if we use what's been mentioned as a guide.

6.4 Userspace Component

Although the userspace component has not reached a point of development where relevant results could be presented in chapter *Evaluation*, it provides a valuable basis for future extension of our work. Next, we will outline how the code contained in directory `userspace` of our source code was formed, in order to review the progress made so far. At the end of the chapter, we will examine problems that arose during the design of the userspace component, so that they can serve as a hint for how to properly extend it.

6.4.1 Design and Implementation

Figure 6.10 shows in an abstract way how the userspace component, in its current form, collects data about the state of the system.

Data collection and decision making about thread placement are done per a predefined time quantum, set equal to 1 second for compatibility with PMWatch's operation constraints. Our view for the state of the system, at least for the time of writing this document, is considered to consist of the following components:

- The utilization of each processing core as extracted from procfs' `stat` file

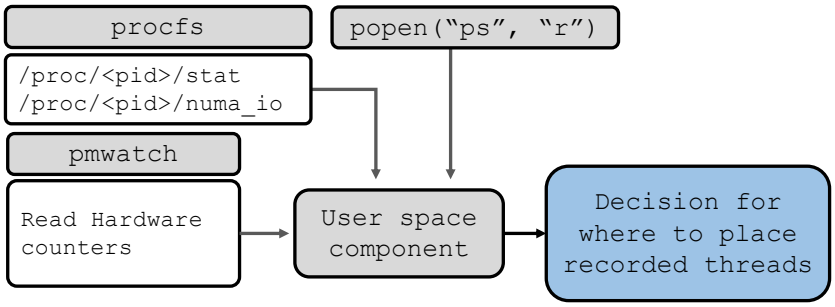

Figure 6.10: Design of the userspace component

- Non-volatile devices' performance information obtained via intel's PMWatch
- A collection of threads resulting from processing the output of program `ps`
- The per-NUMA I/O statistics of each thread, obtained via file `numa_io` described in the previous section

A summary of the thread placement policy based on the information we receive from the system is shown in 6.11. **The policy closely follows the one described in [29]**, as it was considered a good base on which to build the logic to conditionally set the DAX attribute for each file.

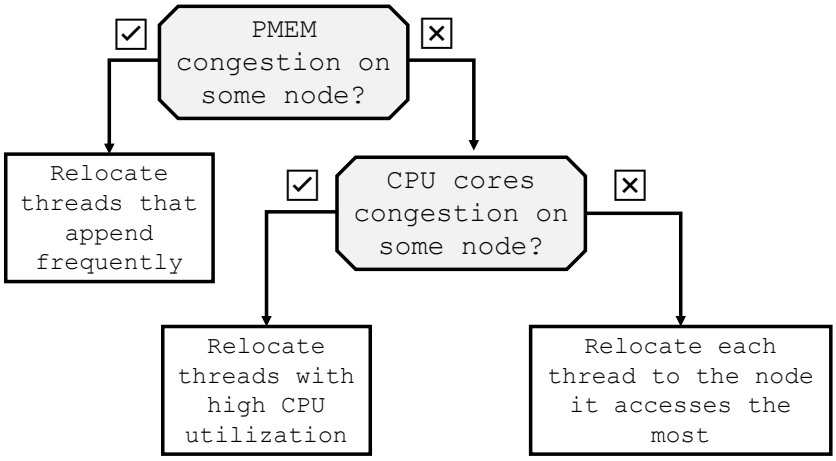

Figure 6.11: Simplified overview of the thread placement logic of the userspace component

It is noted that the very important parameter of data sharing between threads is absent from our logic. If two or more threads frequently write to a shared

file, we want to make sure they are on the same node as the file. Otherwise, significant data traffic will be produced on the interconnection network, with obvious consequences for the performance of accesses to non-volatile memory modules. We will discuss this further in the subsection on how to extend the userspace component.

We then explain the conditions and actions shown in the presented flowchart, and briefly mention how they are implemented in our code.

Non-Volatile Memory Congestion

By the expression "congestion at the PMEM devices of a node," we mean that there is a node for which the total bandwidth (by total we mean that it is representative of both read and write accesses) exceeds a percentage of its maximum possible value, and it is twice the total bandwidth of any other node's. The implementation of how we check this is present in function `nvmm_contention` of `userspace/src/nvmm.c`.

We define the total bandwidth of node n as

$$\texttt{total\_BW}(n) = \texttt{read\_BW}(n) \cdot \frac{1}{3} + \texttt{write\_BW}\ (n)$$

in order to properly normalize it given the fact that the maximum write throughput in non-volatile memories is about three times lower than the maximum read throughput. The maximum write and read bandwidth, as well as the percentage we set as a threshold for congestion, are provided in `userspace/headers/nvmm.h`.

We practically have two ways to get the read and write bytes at each node: in the first way, for each node we sum the relevant counters obtained through the PMWatch API for its non-volatile devices. In the second way, we sum across all threads the node-related fields of the procfs `numa_io` interface. The first way is more accurate in the information it provides compared to the second one, and it can highlight the "write amplification" effect [31]. But this does not necessarily provide the best means of comparison for the purpose of informed thread placement, since the second method gives us a more clear picture of what is done specifically by the threads under the userspace component's management.

Relocate threads that tend to perform multiple appends

When there is congestion of the non-volatile memories, we have two relatively obvious ways to fix it by distributing the load between nodes: the first is to move threads and related data to underutilized nodes. The second is to simply move threads that do a lot of writes that lead to block allocation (called "appends") to underutilized nodes, to create new stored data there. Thus, through

distribution of the newly created data we achieve a distribution of the total accesses.

The first way is beyond the scope of the thesis, but has been studied in literature [13]. Consequently, the second way is the one that makes sense to use in our case. The problem is that we have no way to distinguish the recorded bytes of write accesses into overwrites (writing to already allocated blocks) and appends (allocating new blocks). This could be done by extending the work shown in section *NUMA IO Accounting*, but would require the involvement of the filesystem by its recording special statistics. For now, we are content with the overly simplifying assumption that each thread's writes are appends to a constant percentage, so we use the total bytes written as a hint for how we should choose what threads to relocate.

As we don't want to relocate so many threads to a node that congestion is spawn there, we introduce for each node the concept of usable bandwidth surplus (for the congested node) and correspondingly that of usable bandwidth deficit (for the underutilized nodes). These quantities, which we will denote both as `node_bytes_mean_centered` of node n, are naturally defined in the following way:

$$\texttt{node\_bytes\_mean\_centered}(n) = \texttt{total\_bytes}(n) - \overline{\texttt{total\_bytes}}$$

$$\overline{\texttt{total\_bytes}} = \frac{1}{\#(\texttt{NUMA nodes})} \cdot \sum_{i \in \texttt{NUMA nodes}} \texttt{total\_bytes}(i)$$

Values `total_bytes` correspond to the bytes resulting from the normalized bandwidth at a certain node n, for the elapsed time quantum. Therefore, suppose that in conditions of non-volatile memory congestion we want to relocate a thread from the node under stress to some node n, which thread wrote a total of b bytes during the just elapsed time quantum. We consider that a sufficient condition for avoiding the immediate formation of new congestion, is to move some thread only if the quantity $\texttt{node\_bytes\_mean\_centered}(n) + b$ is not positive.

In the given source code, the logic for moving threads under conditions of non-volatile memory congestion is contained in different functions of `userspace/src/target.c`. These functions contain the string "nvmm." The main function is `fix_congestion`, which implements the provided flowchart.

CPU congestion at a certain node

Parsing file `/proc/<tid>/task/<tid>/stat` that contains the statistics of thread `<tid>` is done via function `cpu_stat` of `userspace/src/cpu.c`. The fields we are interested in are utime and stime, which correspond to the total

time the process has spent in user and kernel mode respectively [26]. These values are measured in kernel clock ticks.

We calculate the CPU utilization percentage of the NUMA node in which the process is executed via function `process_cpu_usage` of `userspace/headers/processes.h`. For ease of reference, we will be mentioning this quantity of node n as $\texttt{cpu\_util}(n)$. The notion of CPU congestion of some node is defined accordingly to the case of non-volatile memories: there is congestion when a node n has CPU utilization greater than a chosen threshold ($\texttt{cpu\_util}(n) > 90\%$), and all other nodes n' have $\texttt{cpu\_util}(n') < 0.5 \cdot \texttt{cpu\_util}(n)$.

Relocating threads with high CPU utilization

It seems intuitively obvious that unintentionally packing some NUMA node with computationally demanding processes can lead to system performance degradation. Therefore, it is appropriate to include the CPU utilization of each thread as a factor when deciding what threads need to be relocated. This has not been apparent in our benchmarks, since they mainly aim to produce I/O traffic, meanwhile the intensity of the calculation done on the input data is not a primary consideration.

Despite efficient load sharing among processing cores having an obvious impact on performance, we recognize from the results presented in chapter *Experimentation* that CPU congestion ends up being of secondary importance compared to congestion of non-volatile memory modules. Related observations have also been recorded in literature [29]. This is tangible if we consider the effects of immediate write throughput saturation, and also rapid performance degradation when remote accesses take place. For this, on one hand there is the relevant ordering between PMEM and CPU congestion checks, as seen in the given flowchart. On the other hand, the same reasoning leads us to proactively checking that future congestion will not emerge for non-volatile memory modules, as seen in the implementation of thread relocation due to CPU congestion in our source code.

The logic for selecting and rejecting target nodes due to CPU congestion is contained in `cpu_preferred_node`, `cpu_target_is_suitable`, and `cpu_decide_target` of file `userspace/src/target.c`. These functions are properly used in function `fix_congestion` of the same file.

Relocating a thread to its most accessed NUMA node

When conditions are ideal, i.e. there is no CPU or PMEM congestion, we wish to move each thread to the node from and to which it is transferring the most

data. This way, we can limit remote accesses and their sometimes destructive effect on performance. This is where the statistics we get from interface `numa_io` of the previous section play an important role. Data collection from this interface is implemented in function `numa_iostat` of `userspace/src/numa.c`.

We would like to limit the phenomenon of a given thread moving pointlessly between nodes (known as "thread oscillation"). This is why we move a thread only if the bandwidth it utilizes from a certain NUMA node is larger compared to any other node by some predefined factor. In our implementation, the logic for selecting the ideal node for a thread is implemented in function `get_target_node` of `userspace/src/target.c`.

6.4.2 More Implementation Details

Everything described here regarding the implementation of the userspace component corresponds to commit (with hash prefix) `7f5fa7d` of the provided repository. The basis for our userspace component implementation was the source code published [14] for the purposes of [24]. In particular, a significant part of the codebase for maintaining a list of selected system threads and updating their information was based on the aforementioned work. This logic can be found in file `userspace/src/processes.c`.

As mentioned, to solve non-volatile memory congestion we want to identify a set of threads of the congested node suitable for relocation, which are doing so ,uch appending that they can cover the bandwidth deficit of the remaining nodes. To do this efficiently, a customized version of the quickselect algorithm was leveraged. Its implementation is in `userspace/src/quickselect.c`, and an associated evaluation routine is available in `userspace/src/testing/quickselect_test.c`. Its logic is implemented in functions `compute_nvmm_status` and `is_write_intensive` of `userspace/src/target.c`.

Although the usage of quickselect helps keep the implementation linear, it must be noted that the most severe scalability issue arises in managing the current threads via a linked list. If the userspace component is suitably extended at some point in the future, it is crucial to use more appropriate data structures, possibly a vector with existence checking and indexing of threads' metadata via hashing.

Another interesting issue is the efficient coverage of the bandwidth deficit in the case of non-volatile memory congestion for systems with more than two nodes. In particular, if we want to make a fair distribution of the utilized bandwidth while trying to respect the locality of threads' accesses, we end up with a problem that turns out to be quite difficult from an algorithmic perspective. In our case, we have two NUMA nodes, so choosing the destination node is triv-

ial. Of course, even though the problem presents particularities in its theoretical formulation, in practice it is most likely that it does not make sense to solve it in the most efficient way, and there may be some satisfactory heuristics for our purposes.

Additionally, we note that there are additional, unmentioned until now, files in our source code related mostly to possible future extensions of our work. One such example is file `userspace/src/dax.c` which deals with setting the DAX attribute of open files. Another case is that of file `userspace/src/hijack.c`, which was created as possible infrastructure for I/O syscall hijacking, mainly for setting the DAX flag of some file before it is first opened. As we will mention in section *More fine-grained DAX support* of the appendix, there has been planning for how to disable the DAX attribute of already open files (i.e. without waiting for its references by processes to reach zero [9]) in order to better support the aforementioned purpose.

To compile the userspace component, we first need to collect the necessary dependencies by running script `userspace/scripts/dependencies.sh` in its directory, and then executing command `make processes` while in directory `userspace/src`. The main dependencies are PMWatch and PMDK, mostly to support a more recent version of the libpmem2 library [25].

6.4.3 Steps to extend our existing design

As a conclusion to this section, we will examine some suggestions for expanding the userspace component as it is in the moment of writing, recognizing some existing shortcomings, and proposing possible methods to address them.

Forming more suitable evaluation infrastructure

As mentioned, part of the reason no evaluation results will be presented for the userspace component is the fact that, at this stage, it presents significant shortcomings. In addition, some experimentation was carried out, but the available evaluation tool, which has been Filebench, was not sufficient to fully examine the efficiency of our userspace component logic. In particular, at least as far as the benchmark's features are concerned, Filebench generates mostly I/O traffic and lacks parameters for performing calculations on the transferred data, thus no really emulating CPU congestion.

Additionally, Filebench's threads have an expected pattern in their behavior, which is identical between threads of the same run. Therefore, when the userspace component acts upon some execution of Filebench, its whole contribution is reduced to just the initial placement of the threads, which is not an inspiring application. Furthermore, each thread randomly selects files during

execution, so threads utilize nodes uniformly, or, equivalently stated, they have no tendency to use any particular node more than the others. Therefore, to effectively examine the userspace component in its current form, a more suiting evaluation infrastructure is needed, either by modifying Filebench appropriately or by using a more complex benchmark, such as RocksDB.

Performing checks for file sharing

The most important shortcoming of the userspace component at the moment is the absence of checking for data sharing between threads. The reason this has not been implemented already is the complexity of properly designing a model that provides a holistic view of file sharing between threads. To be more specific, an appropriate data structure is needed to record the open files of the selected threads, and to identify common open files of theirs. This structure may correspond to a hash table, while it seems necessary to keep tab of open files either by using procfs combined with `fstat` to recover each file's inode, or via syscall hijacking, specifically of `open()` and `close()`.

Even if we have the information about shared open files, we then need to properly define some concept of degree of association between threads, which will be determined by the number of shared open files between threads, and perhaps the frequency with which these files are accessed. Based on this degree of association, we should then determine a thread clustering model that takes into account the need for efficient management of system resources (CPU and non-volatile memory).

Stability Check

Another important aspect for the efficient operation of the userspace component is ensuring its stability. In this case, by the term stability we mean the limitation of thread oscillation phenomena as much as possible. This requires careful analysis of how the decision to relocate threads due to one type of congestion may lead to causation of another type of congestion in the near future, resulting in the direct or indirect relocation of the same thread etc. This turns out to be non-trivial, and some appropriate infrastructure for evaluation is also required.

Better congestion checking condition for non-volatile memory

As mentioned in section *Experimentation*, there are phenomena of sudden saturation or even degradation of the utilized bandwidth of non-volatile memory. Therefore, the condition we have at the moment, which uses as a sole criterion whether we surpass some percentage of the maximum possible bandwidth, is

deemed problematic. A better condition would compare the recorded bandwidth against the expected value based on the number of accessing threads. Generally, we need to define a metric that quantifies how much worse a request's throughput gets because of it coexisting with other pending requests, compared to the scenario of the request being served alone. This way we can be realistic in what to expect in terms of performance. If we are below the bandwidth limit but at the same time have a bad value of the aforementioned metric for many threads of a node (so we have worse utilization than expected given the current condition), it may still be necessary to perform informed thread relocation.

Implementing a congestion check like the one described requires recording the requested size of thread accesses, which can either be implemented via syscall hijacking, or possibly via examining relevant PMWatch fields.

Conditional Use of the Page Cache

DRAM currently offers two important advantages compared to non-volatile memory: its remote accesses do not suffer from the same pathologies, and it offers one order of magnitude better performance. For this reason, the main purpose of creating the userspace component was for it to conditionally include the page cache in the data access path, so as to optimize the overall system performance. A basic condition for this being feasible is the existence of a mechanism in userspace that can inform us on what data is already cached and what data is not, in order to know when there is no use for DAX. For this to be meaningful, it must be done efficiently and without affecting the state of the system, otherwise it would defeat the purpose of the mechanism. This currently seems like a difficult task, and probably needs further investigation into the Linux kernel.

Chapter 7

Evaluation

In this section we will describe the process of evaluating our changes to the ext4 filesystem. The evaluation is obviously useful from the standpoint of examining any working version of our implementation, in order to determine under what conditions we do get the desired performance improvement. However, we also consider evaluation as a valuable tool to assisting the implementation process itself, in order to ensure the correctness and stability of our modifications. By correctness, we mean that the modifications made, ultimately satisfy the desired property of NUMA-locality when allocating data, without altering the logic implemented in vanilla ext4 in any undesirable way. By stability, we mean that the modifications do not violate correctness elsewhere in the codebase in a way that causes the operating system to deadlock, panic, or the file system to destroy its structure.

The evaluation system is exactly the same as described in section *Experimental Setup* of chapter *Experimentation*. Afterwards, we will further analyze the three aspects of the evaluation process (stability, correctness and efficiency).

7.1 Stability Assessment

During kernel development, deadlocks and OS panics were predictably common when testing intermediate versions in a virtual machine. In many cases, the errors in the modifications were immediately apparent either when starting the virtual machine, or when running tasks with FIO or Filebench for long enough. These cases can be detected immediately so as to be debugged. In the general case, however, we have to consider edge cases, such as what happens when the filesystem has few available blocks, or when under conditions of extensive or/and intensive use.

Cases such as those described can be considered to fall under the so-called

aging process of the filesystem. This is of particular interest, especially in evaluating the ability of a filesystem to maintain its performance characteristics over time. For example, this property of non degrading aging was a key pillar in the design of the WineFS [18] filesystem. In its publication, there is heavy use of a tool that simulates the aging process of a filesystem, which is called Geriatrix [19].

Geriatrix is a complex tool that creates, in a randomized way, directories and files within them, based on selected random distributions for aging time, file sizes, and directory structure. Among other things, it also has a parameter for the percentage to which the filesystem may be utilized. For the purposes of this thesis, we do not leverage this tool to evaluate the performance of the modified filesystem, although it is an interesting aspect. We just used it as a means to test its stability under conditions of intensive and exhaustive use. The test was done by calling Geriatrix as follows:

```
$ cd geriatrix/build/
$ ./geriatrix -n 8087252992 -u 0.9 -r 42 -m /mnt/pmem \
       -a ../profiles/agrawal/age_distribution.txt \
       -s ../profiles/agrawal/size_distribution.txt \
       -d ../profiles/agrawal/dir_distribution.txt \
       -x /tmp/age.out -y /tmp/size.out -z /tmp/dir.out \
       -t 1 -i 50 -f 0 -p 0 -c 0 -q 1 -w 2880 -b posix
```

Directory `/mnt/pmem` has been a fixed mountpoint for the non-volatile memory device. In this case we utilized 90% of the available space. After making sure that the run went as expected, we had our first positive indication that we wouldn't attempt to use the modified filesystem on a real system too prematurely.

While testing on a virtual machine, and once on our the evaluation machine, we observed lack of responsiveness after a very short time of using the modified filesystem. The journal did not indicate the cause of the problem, and no useful diagnostics appeared in QEMU's console. The problem is an obvious case of instability. However, its presence was rare and unpredictable enough to make debugging it out of scope within the time window for this thesis. An attempt was made to recreate any conditions causing the problem by utilizing the filesystem extensively, and diagnosing the error early on with GDB, without any success.

7.2 Correctness Assessment

The property on which we structured our changes is reminded to be the NUMA locality of data allocations. Obviously, we need a series of test to check that this property is met, especially for debugging during development. There were many times when the effect on NUMA locality of some

pre-existing optimization was not apparent until the appropriate sanity check was added and led to its detection. Function `numa_test_awareness` of `experiments/scripts/numa/numa_utilities.sh` highlights exactly how the mentioned sanity check was implemented, in its entirety.

7.2.1 Correctness Assessment via FIO

The first and simplest way to check the correctness of our modifications is through a properly configured experiment with FIO. An obvious configuration of such an experiment is to evenly distribute some number of threads across NUMA nodes, each creating and accessing its own file. This is easily done with two jobs, one for each node. The parameter that needs special attention is `create_serialize`. If left at its default value, then the files will be created by the main FIO thread. Therefore, the thread and file placement will be uncorrelated, and the experiment will be meaningless. It is also important that any pre-existing files of this experiment have been deleted.

After the FIO job is completed, we are left to check if the files were placed each on its correct node. It is easy to determine which node each file was intended for, as they are named after the corresponding FIO job. We now want a way to find, in userspace, the node where a specific file was placed.

We take advantage of the convenience of only having 2 NUMA nodes in the evaluation system. We roughly achieve this by finding the first group of the second node via `dumpe2fs`, and comparing it to the group to which the first block of the file belongs, which we can obtain via `debugfs`.

It is understandable that it is not enough to check only the first block of the file, but often its placement is a strong indication of compliance with the property of NUMA locality. Besides, we want the test to be as fast as possible, so that we can proceed promptly with kernel development. The relevant bash code can be viewed in files `experiments/scripts/numa/numa_test_fio.sh` and `experiments/scripts/numa/numa_common.sh`.

7.2.2 Correctness Assessment via Filebench

FIO gives us a direct way to check the correctness of our modifications. However, it does not have the necessary degree of complexity in its accesses, nor a dynamic element to the way it places threads, as to make sure that the NUMA-locality property of data allocation persists in more complex tasks. This is exactly the problem that the Filebench tool can solve. As we mentioned in chapter *Methodology*, this tool can simulate the behavior of different types of programs in terms of how they perform I/O.

The most important feature of Filebench in this case is the fact that it defines a set of files, some of which are preallocated, while some others are allocated at execution time if randomly selected. This randomness in how files are created can lead to the formation of a test that will better certify us the robustness of our proposed modifications.

Filebench itself does not contain the concept of NUMA topology in its design, meaning that there is no way to express preference in how spawned threads may be placed. Also, remember that in order to check afterwards whether data allocations have been as expected, we must define a mapping between the name of each file and the node on which we want it placed.

For the mentioned reasons, we slightly modify Filebench's source code so that upon file creation, the execution of the thread is limited to the node indicated by the remainder of the file's identifier when divided by the total number of nodes. For example, having two nodes, files whose name matches an even number are placed at node 0, and vice versa. The necessary modifications are present in `patches/filebench_test.patch`, which must be applied to Filebench's source code before any relevant test can be performed.

The correctness assessment routine via Filebench is available in `experiments/scripts/numa/numa_test_filebench.sh`. Executing it reveals that the file placement error percentage drops from 50% for vanilla ext4, to less than 5% for the modified version. Certainly, the fact that the percentage does not end up being zero is discouraging, but it is considered to be within the bounds of what is permissible given that we are trying to qualitatively demonstrate the effect of NUMA locality on performance, and is justified by the increased complexity of ext4's codebase.

7.2.3 Correctness Assessment Solely via Bash Script

This test is quite similar to that of FIO, with the difference that the creation of the files is done directly in our bash script through the redirection operator, and command `taskset` to select the corresponding node. In general, creating our test files looks like this:

```
$ taskset -a -c <cpu_id> bash -c "./scripts/bigfile.sh >
    ↪ /mnt/pmem/file_<cpu_id>_\$i"
```

We would expect that if the FIO test succeeds, so would the Bash-only test. Surprisingly this does not seem to be the case, at least not always. The investigation of this issue was carried out in appendix *Investigating the behavior of the Bash shell redirection operator*.

This case is interesting, mainly because it presents the need for investigating

when the unavailability of suitable kernel workers can be destructive to NUMA locality. Checks made via FIO on a working directory having DAX disabled, have shown that under normal write conditions, i.e. via syscalls to VFS, there is one kernel worker for each NUMA node. Conclusively, the source of the problem we identified with the aforementioned investigation is bypassed, and the soundness of our design in this case is not affected.

Another interesting fact is that in early stages of ext4 modifications, this quirk of the Bash-only test was not apparent. This was because the target NUMA node was chosen based on each file's inode, for debugging ease. That is, the node on which the file's metadata was allocated was the one on which all accesses took place going forward, instead of the node hosting the appending thread. So, as long as the metadata allocation was correct, any appends after that were performed to the same node, and the test appeared to be successful. This highlights the importance of carefully formulating and developing our evaluation methodology.

7.3 Performance Assessment

Having verified the correctness and stability of the modifications made to ext4, it's time to examine to what extend the performance issues presented in section *Experimentation* have been fixed. We'll first look at what speedup we get for simple microbenchmarks via FIO, and then we'll perform an evaluation for more complex tasks through Filebenh. In all cases, the metric we're interested in is bandwidth.

It's very important to mention that switching between vanilla and modified ext4 for the purposes of benchmarking is done via the mount parameter `numa` we added. Extensive effort was made to form the changes to have zero effect when mount parameter `numa` is not given, i.e. to have equivalent behavior to the original version of ext4. However, the correct thing to do would be to have a separate filesystem with our modifications, just as it was done with the NOVA filesystem for the WineFS publication [22]. However, due to lack of time, and based on some informal tests that showed the `numa` mount parameter to be working as expected, the alternative was preferred.

We first mention the possible different configurations of devices and the filesystem which we will use for our experimentation. Epigrammatically, we have:

- **Single device**: We use only one device, e.g. `/dev/pmem0`, and the original version of ext4.

- **Linear map.**: We concatenate devices `/dev/pmem0` and `/dev/pmem1` via the device mapper, and format the linear device with the original version of ext4.
- **Striped map.**: We interleave chunks of devices `/dev/pmem0` and `/dev/pmem1` via the device mapper, with chunk size equal to a page's size, namely 4KB. On top of a striped mapping, it only makes sense to use the original version of ext4, which we do. We recall that the modifications made to ext4 are based on the assumption of having the devices linearly concatenated.
- **Linear map. (modified)**: We have a linear concatenation of devices `/dev/pmem0` and `/dev/pmem1`, and use the modified version of ext4 for the linear device. This layout is the only one representative of our work.

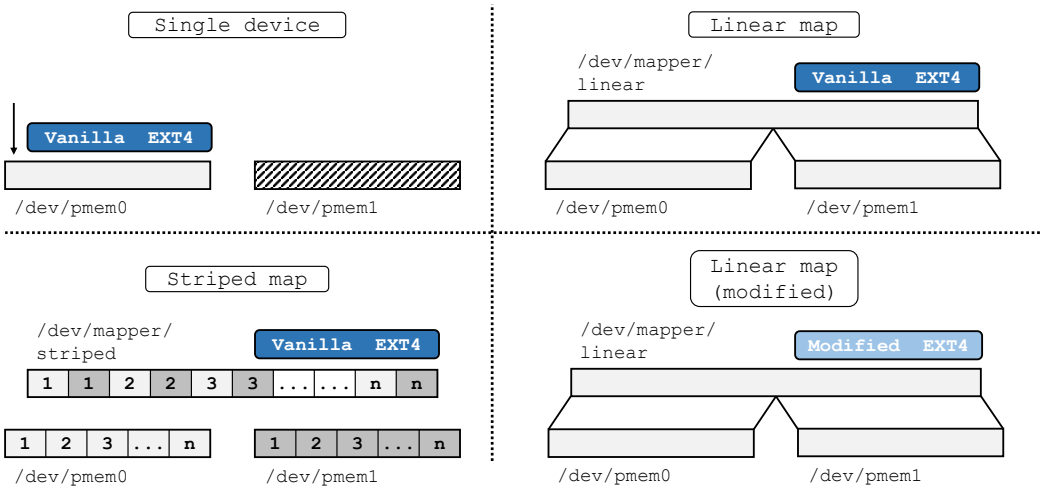

Figure 7.1: The different configurations used for evaluation

7.3.1 Performance Assessment via FIO

We construct a simple experiment via FIO, in which we distribute threads evenly among NUMA nodes, and each thread works on its own 256MB file, accessing it with 0%, 50%, or 100% read percentage. We have the DAX feature enabled. The FIO parameters we focus on are the number of threads per node and the percentage of accesses that read, which is the same across threads. We present the results of conducting the experiment in a Linear map. (modified) configuration, normalized to the results of the Linear map configuration. Thus, we intend to highlight the performance gain brought about by the

work presented in chapter *Implementation*. The experiment is described in file `experiments/scenarios/test_mods.sh`.

It is noted that whether we use the Linear map. or Single device configuration as a normalization base, we get qualitatively the same results, since the total size of the experiment's files is comfortably covered by the size of a single device. Also, to make the experiment meaningful, we make sure that FIO's `create_serialize` parameter is properly set to 0.

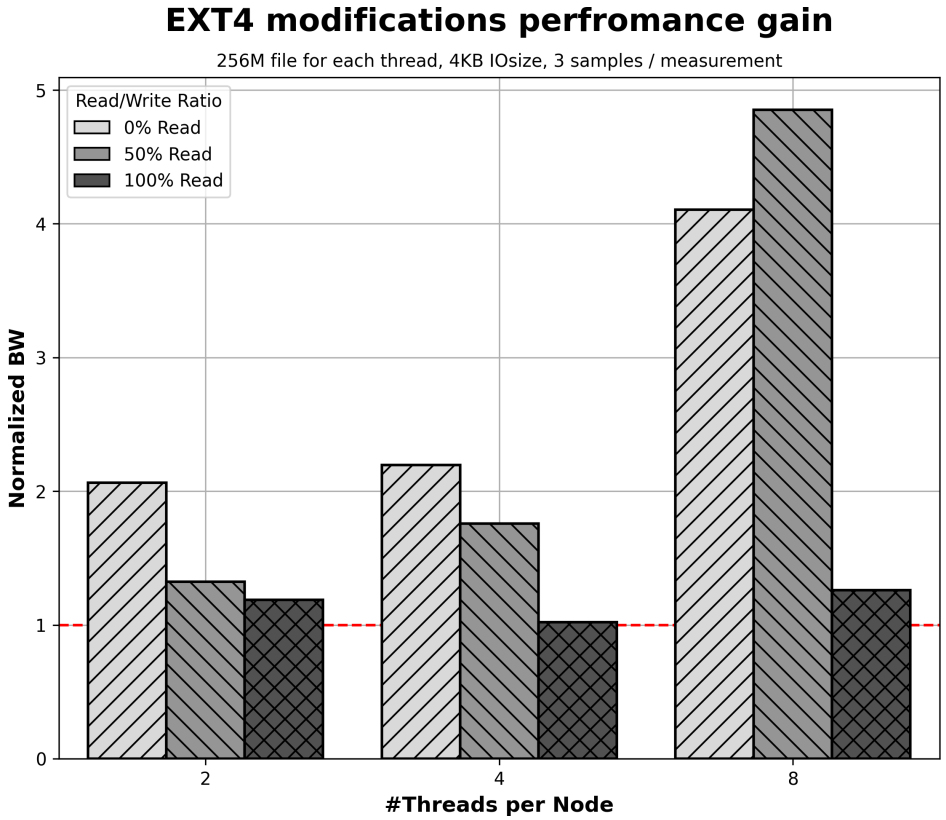

Figure 7.2: Evaluation results via FIO (up to 8 threads per NUMA node)

Figure 7.2 shows the results we get from the experiment. The first conclusion we draw is that pure writes benefit directly from the NUMA locality property. On the other hand, pure reads have an expectedly negligible performance gain, since the changes we made do not directly affect this type of access.

What is particularly impressive is the speedup we get for mixed accesses, which also tends to be greater than the speedup we get for pure writes. This is due to combating the "Read After Remote Write" phenomenon, which we achieve by allocating locally. Indeed, as shown in figure 7.3, in which we have included measurements for 16 threads per node, the improvement appears to be remark-

able, and our design finally manages to address the noted pathologies of non-volatile memories on NUMA systems .

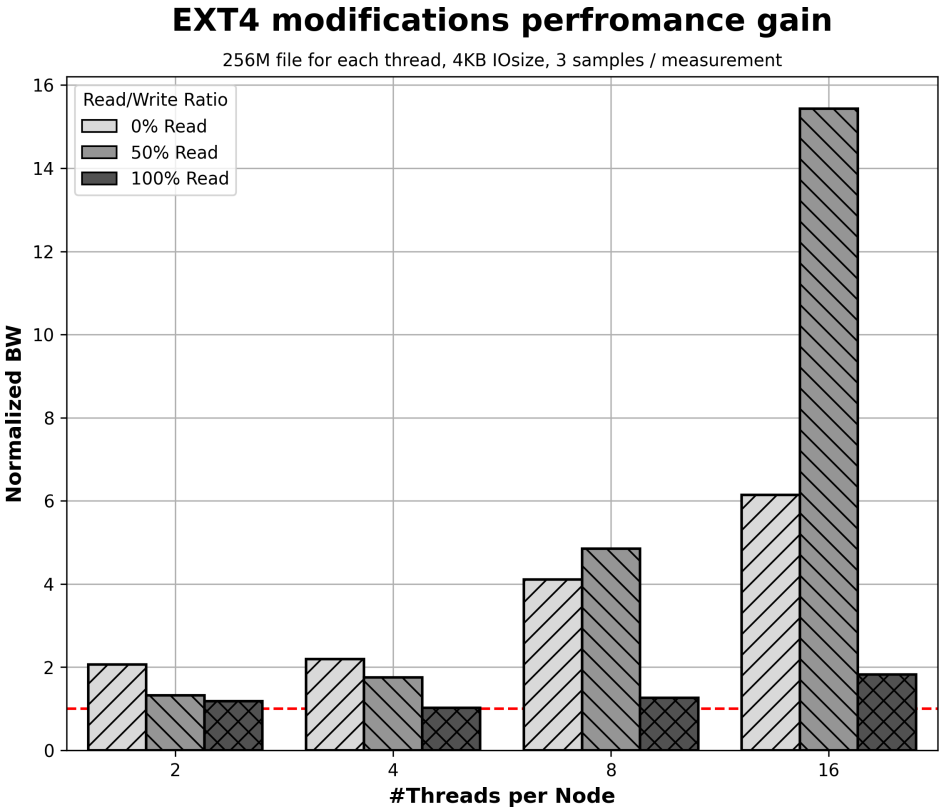

Figure 7.3: Evaluation results via FIO (up to 16 threads per NUMA node)

7.3.2 Performance Assessment via Filebench

In this section, we wish to examine the behavior of the modified file system in more complex workloads. The Filebench benchmarking tool is ideal for this purpose.

As previously mentioned, Filebench does not include the concept of NUMA topology in its design. This makes a lot of sense, since its purpose is to be a tool for examining the performance of basic interfaces offered by filesystems. The modified filesystem of this work takes as a data allocation hint the node from which the corresponding thread is executed, which is admittedly a form of special behavior. Therefore, Filebench has no parameters for choosing a node to preallocate files to, nor for which node to place execution threads on.

Additions to Filebench

In order to extend the capabilities of the tool for the purposes of our work, a number of additions were made which are recorded in file `experiments/patches/filebench_numa.patch`. Following is the reasoning behind the changes, and what they add to the functionality of Filebench.

Preallocating files is intended to create a state that the program could have reached from some elapsed execution interval. We consider that a thread tends to run more on a specific node, which is very likely, either due to the use of `taskset` to better utilize lower level Caches on each node, or given the use of software like the userspace component we described. Therefore, in the hypothetical preceding execution interval, the fileset of a thread will tend to be concentrated on one node.

Based on what has been said, we understand that it makes sense to add an optional `numa` parameter to Filebench filesets, which takes an integer value corresponding to a node ID. When said parameter is absent, we have the same behavior as that of vanilla Filebench. Given this, we place the preallocating thread on the specified node in order to gather the fileset files there.

We also add a similar optional parameter `numa` for the "thread" field of a process, which specifies the NUMA node in which the threads defined by the field are placed. This parameter makes sense as well, as we want to look at the performance impact that targeted thread placement can have. This would be indicative of the capabilities of auxiliary software infrastructure, like the userspace component we described.

Configuring and Defining Workloads

The adjustments made to the workloads to run on our machine are contained in file `experiments/patches/filebench_workloads.patch`. The main difference compared to the preexisting workloads is that we specified a DAX enabled directory of our modified filesystem as a working directory (usually this directory is `/mnt/pmem/daxdir`), and that we increased the total number of files. In `varmail` specifically, we increased the average file append size to 256KB, so that the effect of NUMA aware allocations may be made clearer with this example. This is not unreasonable, as, for example, attachments to mails other than plain text could also be transferred through the server.

Two new workloads have been added, `fileserver_numa` and `webserver_numa`, which work exactly like `fileserver` and `webserver` with the difference that for each workload two instances are run in parallel, with each instance having its own fileset and running on a different NUMA node. We will use work-

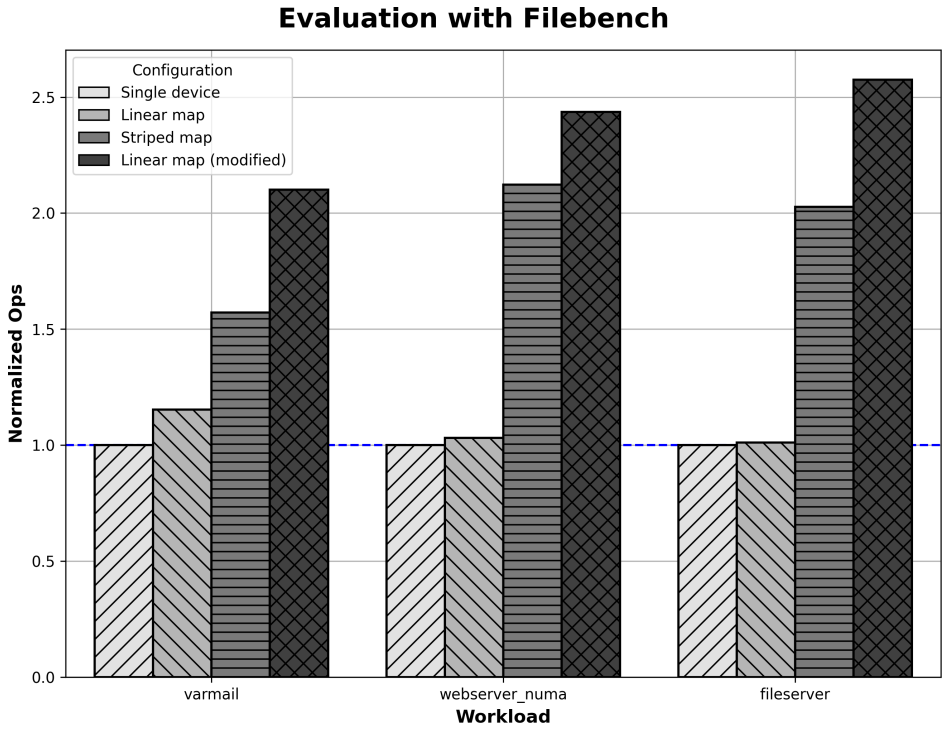

Figure 7.4: Results of our evaluation via Filebench

load `weberserver_numa` instead of workload `webserver` itself, as it is focused on read accesses, which modified ext4 only benefits from when the file to be read has been allocated in a NUMA aware manner. Otherwise, we would not observe any significant variations between the different execution configurations.

Figure 7.4 shows the results from executing the experiment of file `experiments/scenarios/filebench.sh`. Also, table **??** lists the percentage increases in the performance of Linear map. (modified) configuration compared to Striped map. and Linear map.

Workload	**Improvement over Linear map. (%)**	**Improvement over Striped map. (%)**
varmail	82	34
webserver_numa	136	14
fileserver	154	27

Table 7.1: Performance improvement on selected Filebench workloads

Analysis of our Results

We see that the configuration representing the modified version of ext4 is deemed to provide the best performance in every case. The performances of the Single device and Linear map. configurations are slightly different from each other, because one device ends up being used much more than the other, due to the size and number of files in the Linear map.

Striped map. has comparable performance to Linear map. (modified). The reason we have included it as a configuration, is that Striped map. is something that can be setup directly, without the work of this thesis. Also, it has the prospect of working better than the Single device and Linear map. layouts, because it can and does expose at the level of individual accesses the aggregate bandwidth of all devices, regardless of whether, as we will mention, this ends up underutilized. Therefore, we have to compare this work with any pre-existing, easy and relatively efficient solution.

`Varmail` demonstrates the comparative advantage of modified ext4 for large file appends, which appends represent responses to messages. We also get from this workload the greatest differentiation compared to configuration Striped map, precisely because of the aforementioned advantage. However, there doesn't seem to be anything in the pattern of I/O that differentiates the performance of each configuration when not taking into account file appends. Therefore we believe that the decisive factor in the performance gain is almost exclusively appending to files.

For `weberver_numa`, we benefit greatly from modifications. Interestingly, the difference with the Striped map. layout is small, but not negligible. For the latter, we get some performance boost because we have files solely being read, and the simultaneous access to multiple NUMA nodes in this case mostly exposes the aggregate bandwidth of the devices, rather than having a destructive effect as we would get to expect at this point. This makes sense from an overview of figure 4.1, where it can be seen that if we only have read accesses, the bandwidth of local accesses differs not that much compared to that of remote accesses. Where the modified layout gets an advantage is due to appending to the log file.

Finally, for `fileserver` the biggest performance gain is observed in relation to Signle device and Linear map. configurations. It also differs strongly from the Striped map, and a non-negligible factor of this seems to be the fast retrieval of metadata sought by command `stat`, which is achieved due to NUMA locality in metadata accesses being part of our design.

Further Analysis

What is of most interest is the potential effect that the distribution of blocks of a file across nodes may have. This situation may happen due to appending to the same file either by distinct or moving threads. The results so far do not show this effect, however it would be interesting to develop some microbenchmark that aims to analyze it.

Additionally, the relative placement of a file's metadata and data on different nodes matters, and how this can affect performance. In `varmail`, after intensive parameterization during experimentation (which unfortunately was not sufficiently documented to be reproduced), severe performance degradation was observed in any configuration that includes more than one node. The main reason was suspected to be metadata synchronization via the `fsync` function, which causes multiple small remote write requests, and possibly leads to the "Read After Remote Write" problem reappearing. Therefore, it would be of great interest to consider an efficient metadata synchronization scheme in a future extension of the work.

Finally, although Striped map. appears to be approximately as good a solution as our modified configuration, in reality its performance is bounded in a sense. That is, although it utilizes the bandwidth of multiple devices simultaneously, it does so in a way that always enforces uniform access to the devices, and any remote access pathologies cannot be controlled. Therefore, we can think of it as an underutilization of aggregate bandwidth. The modified filesystem maintains a clear separation between devices, and informed placement of execution threads can lead to spectacular gain, as shown in figure 7.5, for which workload `fileserver_numa` is included .

The speedup is so great because workload `fileserver` defines 50 threads, which seem to far exceed the saturation point of each node's memory bandwidth. By having a clear separation between the two parallel processes of workload `fileserver_numa`, we seem to be on the way to fixing multiple performance bottlenecks. However, this phenomenon is more pronounced due to the distribution of the threads of each process in separate nodes, which highlights the need for further developing the userspace component.

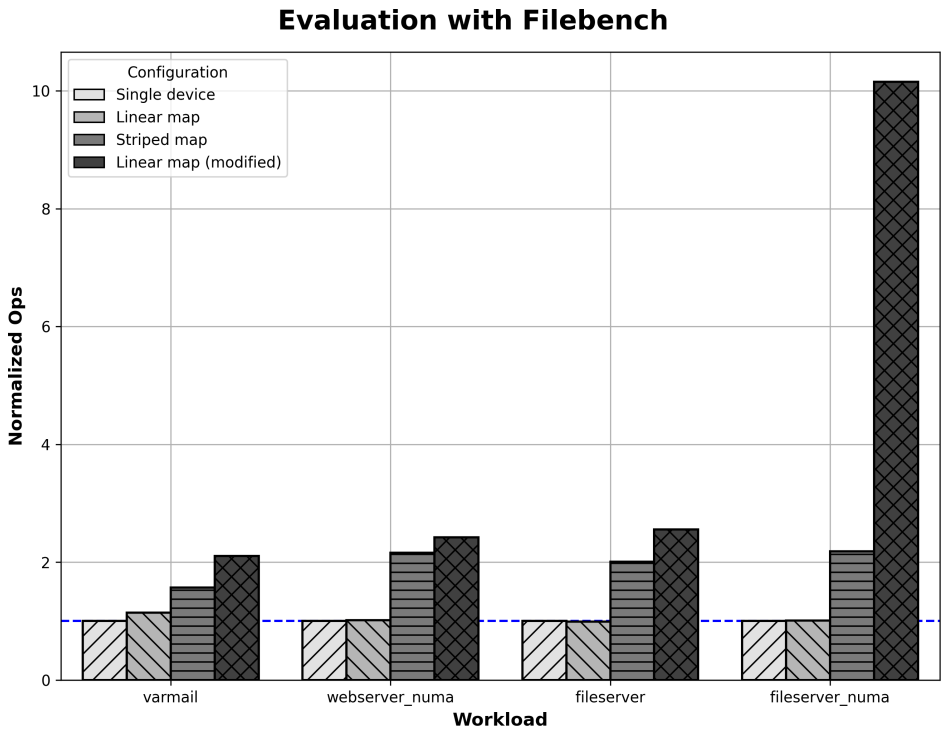

Figure 7.5: Results of our evaluation via Filebench (including workload `fileserver_numa`)

Chapter 8

Related Work

Between the announcement of Intel's non-volatile memory modules and their commercial availability, researchers relied on what information was publicly available about the behavior of this product, carrying out their work facilitated by simulation systems for the purposes of evaluation [31, 30]. An important product of this period is the NOVA filesystem [30], which adapted log-structuring techniques (serial writing of (meta)data in circular buffers) in a way that takes better advantage of fast random access as provided by PMEM, compared to conventional filesystems.

Several research projects [29, 22] have extended, in a makeshift way, the NOVA filesystem to support allocating over multiple NUMA nodes.

Later came WineFS [18], a filesystem that extended Intel's PMFS so that allocations would preserve the system's ability to serve hugepages even under conditions of significant aging. This remarkably improves performance for memory-mapped file accesses.

Regarding the use of non-volatile memory in NUMA systems, the literature is limited. WineFS supports creating a filesystem instance on multiple NUMA nodes. WineFS' NUMA awareness logic is based on uniquely matching processes and nodes, and moving the process to the selected node every time it wants to perform a write access.

The case of NapFS [17], a "meta-filesystem" for providing NUMA awareness, is of special interest. It establishes a selected filesystem per NUMA node, and forms additional software infrastructure on top of that, including auxiliary threads and global data structures, to enable allocations on multiple nodes in an interleaved fashion.

Additionally, there has been research seeking to highlight the key performance characteristics of non-volatile memory modules. One of them [31] has been used extensively in literature, and has evidently been a valuable resource in understanding the qualitative characteristics of said modules for the purposes of

this work. Other research [16] similarly analyzes the fundamental performance characteristics of these devices, both as storage media and memory.

Chapter 9

Epilogue

9.1 Conclusions

Our purpose with this thesis has been to examine the peculiarities of current non-volatile memory technology in regard to non-uniform memory accesses, and how we can properly adapt ext4's (meta)data allocation routines to better respect any noted intricacies. Indeed, we have seen from our experimental evaluation that such an approach can ultimately lead to significant gain in how well we utilize the total available bandwidth of a system's non-volatile memory devices.

More generally, we re-demonstrated that the strategy of uniformly using some NUMA system's resources may not be the optimal way to make good use of them, at least it is deemed less efficient than a strategy which takes into account a clear separation between NUMA nodes and works on the most suitable one on a case-by-case basis. In terms of work presented in this document, the first strategy corresponds to the unification of all non-volatile memory devices into a striped mapping, and the second strategy corresponds to the design presented in section *Implementation*.

As mentioned in various parts of this thesis, the work documented allows for a number of interesting extensions, which will be mentioned (mostly briefly) in the next section.

9.2 Possible Extensions of Our Work

The most immediate extension is continuing the development of the userspace component, in what ways have been described in subsection *Steps to extend our existing design* of the relevant section in chapter *Implementation*. Regarding the ext4 filesystem on the other hand, we can first add support for

the `flex_bg` feature to our NUMA awareness logic. Also, we can examine the effect of our modifications on how well the filesystem ages. In particular, it is necessary to examine in more detail the performance penalty when there is an arbitrary distribution of a file's blocks between NUMA nodes. This can very probably be the result of appending by multiple threads on different nodes.

Afterwards, we present in more detail two of the most important, and potentially complex, possible extensions.

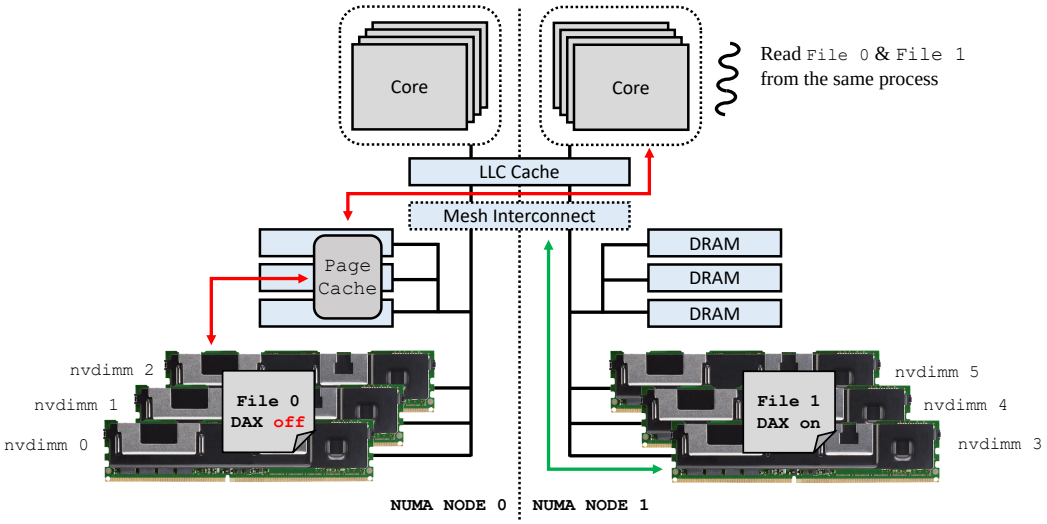

Figure 9.1: Possible way of using the per-file DAX feature to smooth the effect of remote accesses

9.2.1 Smoothing the effect of remote accesses via the per-file DAX feature

As previously mentioned, one of the main reasons ext4 was chosen to be extended, is the control it provides over the DAX feature, most remarkably on a per file basis. Given this, we can form some method with which we selectively use either the page cache or the DAX feature for different files, in order to better utilize the cumulative bandwidth of both non-volatile memories and RAM. Such a system would assign files with relatively high degree of accesses' repeatability to the page cache, since in such way we take advantage of the wider bandwidth of volatile devices, making worthwhile the cost of copying pages to the page cache.

The mentioned method could be adapted to encourage accessing files through the page cache when they receive frequent remote accesses, in order to avoid the pathologies mentioned in this thesis. However, for this extension to make sense,

one must first carefully consider how direct access and page cache operations may interact with each other, with a focus on how writeback to non-volatile devices can have a similar negative effect to the "Read After Remote Write" pathology.

9.2.2 Local Overwrite Support

In this work, we have partially succeeded in directing new block allocations as to better respect the property of NUMA locality. Obviously though, writing to already allocated blocks can only be done on the NUMA node to which the blocks correspond. If the data to be written is large enough, we could write the modified pages temporarily to a block on the local node and then adjust the extent tree to point to the modified pages instead of the original ones on the remote node.

Supporting such a feature has many design parameters to consider. A first example is the need to efficiently transfer to the local node the pages to be modified, when necessary. Next, we need to take a closer look at how the extent tree works, to understand in what ways we may modify or fragment the extents to be updated without running into synchronization issues. When we refer to fragmentation, we mean that if we write to some contiguous clusters of a large extent, we would not want to transfer the entire extent to the local node in order to do the writing there. Instead, we would prefer to split it into (two or three in some cases) sub-extents so that one of them contains the clusters we want to update, and only transfer what is necessary.

So it's understandable that the proposed extension presents challenges, but it can be very performant for large file updates, especially if the file is accessed in write-only mode, where we might be able to skip transferring some pages. But one first needs to quantify the cost of transferring pages and splitting the extent tree versus simply performing the remote access. A good starting point for experimenting with ext4's codebase is the `if` structure under the `"Lookup extent status tree firstly"` comment of function `ext4_map_blocks`.

Chapter 10

Appendix

10.1 Investigation of the "Read After Remote Write" Effect

This section will present the effort made to determine the cause of the "Read After Remote Write" phenomenon. Although the investigation was not conclusive, it is recorded here in order to provide a basis for continuing this effort at some point in the future. For convenience, we describe the problem again:

Problem Description

- We have multiple remote threads, each with its own file, accessed via DAX.
- Writing to said files has been the immediately previous action.
- Each thread tries to read its file, but it uses significantly less bandwidth than expected the first two times.
- In reading the files a third time, we observe a normal measurement for the utilized bandwidth.

In order to be sure that the phenomenon is not caused by some parameter of FIO that we cannot know due to the tool's complexity, we create a very simplified version of it from scratch. We call the simplified tool cfio (custom FIO), and it can be found in file `various/cfio.c` of our source code. The program has the following parameters:

num_threads <int>	:	Number of threads (each processing its own file)
directory <str>	:	Where threads find/create files
job_name <str>	:	Name the files have received / will receive (of the form `directory/job_name_<thread_number>`)
file_size <long>	:	Size of files in bytes (useful only if files do not already exist and need to be created)
block_size <int>	:	Size of the write/read buffer in bytes
loops <int>	:	Number of times we traverse each file
invalidate <bool>	:	Flush the relevant files from page cache before execution (Default: False)
is_write <bool>	:	Write to the files instead of reading (Default: Read files)

We define an execution scenario in which 16 remote threads (remote in the sense that we use the devices of a single NUMA node and place our threads on another) first write to their own file, and then read their files three times:

run_cfio.sh

```
#!/bin/bash

dir=$1
job = JOB
n=16
cpuset="16-31" # Remote access

# Write to the fileset
taskset -a -c $cpuset ./cfio --num_threads=$n --directory=$dir
    ↪ --job_name=$job --block_size=4096 --loops=1 --is_write

# Read the fileset
for i in {1..3}; do
taskset -a -c $cpuset ./cfio --num_threads=$n --directory=$dir
    ↪ --job_name=$job --block_size=4096 --loops=1
done
```

We get the following output:

```
$ ./run_cfio.sh /mnt/pmem0/phtof/daxdir
BW = 153.868 (Kbps)   # Write
BW = 1,694,802 (Kbps)   # Read 1
BW = 3,450,831 (Kbps)   # Read 2
BW = 14,765,409 (Kbps)  # Read 3
```

We see that it takes 3 executions to get a reasonable value for the read throughput. Since we have confirmed that the problem does not arise from the

complexity of FIO's implementation, we proceed with our investigation.

10.1.1 Operating System Behavior Investigation

We use flamegraphs [7] to interpret the perf log to see if maybe some different functions are called between problematic and non-problematic reads, getting the results shown in figures 10.1 and 10.2.

Figure 10.1: The resulting Flamegraph for the first set of read accesses

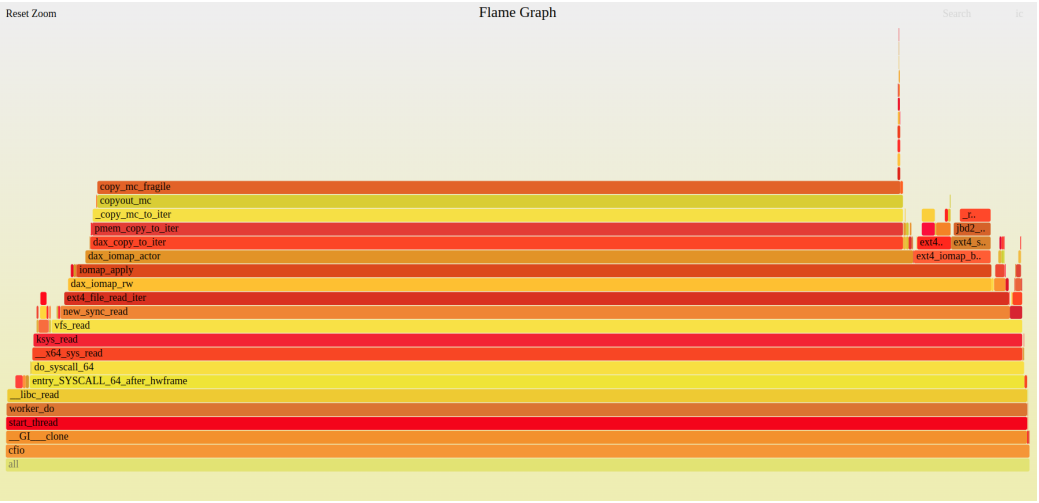

Figure 10.2: The resulting Flamegraph for the third series of read accesses

Although at first glance the two graphs look different, they actually have quite similar samples. In the first one (corresponding to the slowest execution), functions that are time-consuming (like `copy_mc_fragile`) have so many samples that they overshadow samples associated with preparation functions for the more essential functions. It is these initialization functions that show up more

prominently in the second graph (like `ext4_iomap_begin`). That is, on closer inspection, they do not seem to differ in "which functions are executed," but in "how long certain functions are executed."

At some later point in time, the same measurements were performed for the NOVA and WineFS filesystems (in this work, it is always implied that ext4 is used, unless explicitly stated otherwise). The results are logged in files `experiments/measurements/nova_4G_read_after_write.json` and `experiments/measurements/winefs_4G_read_after_write.json`. However, their behavior was similar to that of ext4, even though these particular filesystems are specifically built for increased performance when running over non-volatile memories.

Let's assume for now that there isn't something going on at the OS level that is causing the difference in performance of the different sets of reads. Of course, reviewing findings such as those of FirePerf [20], it is evident that this assumption can possibly be ill-founded.

10.1.2 Investigating the behavior of non-volatile memory devices

We use PMWatch to get device usage statistics while running our benchmarks. First, we do a series of allocations. Then we start PMWatch with a period of 1 second, run it in the background, start executing a series of read accesses, wait 1 second to get at least one sample of PMWatch, and finally stop its execution.

```
$ ./run_cfio_once.sh 1 # Remote write
$ sudo pmwatch 1 > 1.txt & ./run_cfio_once.sh 0 ; sleep 1;
    ↪ sudo pmwatch-stop
$ sudo pmwatch 1 > 2.txt & ./run_cfio_once.sh 0 ; sleep 1;
    ↪ sudo pmwatch-stop
$ sudo pmwatch 1 > 3.txt & ./run_cfio_once.sh 0 ; sleep 1;
    ↪ sudo pmwatch-stop
```

Script `run_cfio_once.sh` is similar to `run_cfio.sh`, except that the directory is predefined instead of being a parameter, and now the only parameter to the script is a binary value that is equal to 0 when we wish to perform read accesses, or equal to 1 otherwise (for pure write accesses).

In figure 10.3 we see part of the data we got for the first DIMM (as long as the DIMMs of a node are interleaved, whichever we see will probably give a representative picture). We notice that although we are reading files, the device shows for the first series of reads that there are also writes performed by the processor (of the same order order of magnitude - in bytes - as the write accesses).

This ceases to be true in the third series of reads, for which it appears no writing to be done at all.

	A	B	C	D	E	F	G	H	I	J	K	L
1	timestamp		DIMM0									
2	epoch	timestamp	bytes_read (derived)	bytes_written (derived)	read_hit_ratio (derived)	write_hit_ratio (derived)	media_read_ops (derived)	media_write_ops (derived)	read_64B_ops_received	write_64B_ops_received	cpu_read_ops	cpu_write_ops
3	Εκτέλεση 1											
4	1677755101	2100104720	65477552	56669440	0.73	0.75	2557717	2174490	18928828	8697960	9310560	8629929
5	1677755102	2100103522	69541785	6 01703168	0.73	0.75	2716476	2350403	20267516	9401612	9972331	9328001
6	1677755103	2100102762	53248	799488	0	0	208	3123	13324	12492	38	17
7	Εκτέλεση 3											
8	1677755123	2100100714	53760	707072	0	0	210	2762	11888	11048	39	17
9	1677755124	2100101240	56832	802560	0	0	222	3135	13428	12540	37	17

Figure 10.3: Study of the first and third series of reads via Intel's PMWatch

An additional test was performed, with which we waited for some substantial interval between writes and the first set of reads in case something was done internally by the devices after the write accesses, therefore affecting the reads, but the result is exactly the same regardless of how long we wait between writes and reads.

10.1.3 Investigation at the level of system architecture

We see that the processor sends write operations to the device for no apparent reason. The first assumption made is that maybe this behavior is related to the iMC: Maybe in some WPQ (write wait queue) something is happening and the writes to the device are not completed, so data is just sitting on the ADR without being written to the storage medium. However, it is difficult to write such a large amount of data in the first series of reads: There does not seem to be any buffer structure, that we know of, in the path between the storage medium and the processor to have such a large data capacity.

In the process of searching whether this phenomenon has been observed elsewhere, a related document [1] was found with a summary report of problems that Intel has observed for the 2nd generation of Xeon processors. From the problems listed, the following seemingly relevant titles were noted while quickly going through the list. It must be specified, however, that many of the mentioned technical terms were not obvious, and therefore the choices may be incorrect with a high probability. The only reason most cases are noted is because they mention some MSRs, which might may be interesting to examine in a possible extension of our investigation.

> The next two problems, although they refer to non-volatile memories, do not concern us since the memories in our experiments were set only in App Direct Mode.

- **Intel Optane Persistent Memory Mode or Mixed Mode May Cause a Hang**

When the processor is utilizing Intel Optane Persistent Memory Mode or Mixed Mode, a Core MMIO read request may incorrectly block a write request from the IO subsystem, leading to a system hang.

- **Systems Using Intel Optane Persistent Memory in Mixed Mode May Experience a System Hang or Reset**
  When the processor is utilizing Intel Optane persistent memory in Mixed Mode with both App Direct and Memory Mode Snoopy Modes enabled, a system hang or reset may occur when running a workload.

The next three problems appear to be related, but are either reported to lead to hangs rather than simple malfunctions, or involve the operation of a hardware prefetcher for which it is unspecified under what conditions it behaves unexpectedly.

- **Memory Controller May Hang While in Virtual Lockstep**
  Under complex micro architectural conditions, a memory controller that is in Virtual Lockstep (VLS) may hang on a partial write transaction.

- **System May Hang When The Processor is in 3 Strike Due an Internal Mesh-to-mem Error**
  Under complex micro architectural conditions, the processor may hang with an error mesh-to-mem caused by a core 3-strike with Machine Check Exception (MSCOD=80h, MCACOD=0400h) logged into IA32_MC3_STATUS (MSR 40Dh).

- **XPT Prefetcher May Not Perform as Expected**
  When XPT prefetcher is enabled it may not prefetch as expected on memory channels that contain Intel Optane Persistent Memory.

The next problem is the one we think deserves some investigation, as it seems the most relevant and it mentions not only a possible system hang, but also "unpredictable system behavior," which is encouraging. In addition, it explicitly mentions the case where we have more than one socket

- **WBINVD/INVD Execution May Result in Unpredictable System Behavior**
  Under complex micro architectural conditions, the processor may hang or exhibit unpredictable system behavior during Writeback and Invalidate Cache (WBINVD) or Invalidate Internal Caches (INVD) cache instruction execution on a two or more socket system.

10.1.4 Possible extensions of our investigation

In an attempt to further extend the investigation presented in this section, some of the possible approaches are as follows:

- Read several hardware counters via perf stat and PMEM related names or codes as documented by intel [2]. It is quite a difficult process though, and many times the description given for the counters is not particularly informative.
- Possible tracking of MSR registers, like some of those mentioned in the problems listed earlier.
- Find a methodology to check if flushing just allocated data has been performed normally, or if there is anything pending. For example, we would make sure there are files that contain a certain recognizable pattern (e.g. AAA...), modify cfio to write a different, distinct pattern from the previous one (e.g. BBB...), and to proceed with writing it to the files. Next, we'd like to unmount the device without encouraging further flushing, if possible, checking the contents of the device to see how far the old pattern has been replaced.

  This methodology has several peculiarities that make it probably infeasible, mainly because the umount command, or reading while we have the device attached, is very likely to lead to a data flush, and the observation itself will ultimately affect our experiment.

10.2 Investigating the behavior of the Bash shell redirection operator

As mentioned in the main body of this document, the modifications made to ext4 do not achieve NUMA locality when we attempt to allocate files via bash's redirect operator, even if we have used `taskset` to ensure that writes are performed to a specific NUMA node. This is interesting, because equivalent tests (e.g. via FIO) are successful. To investigate this finding, we employ the method of remote debugging of a QEMU virtual machine via GDB. We set the next breakpoint in GDB:

```
break ext4_mb_new_blocks if (ar->inode->i_sb->s_mount_opt2 &
    ↪ 0x00000100)
```

This breakpoint stops VM execution when function `ext4_mb_new_blocks` is reached, only when parameter `numa` (corresponding to mask 0x00000100) has been provided as a mount option, so that the breakpoint is not triggered repeatedly due to actions for the root directory's ext4 instance. Running our test in `experiments/scripts/numa/numa_test_redirection.sh` triggers the breakpoint. We get the following overview with GDB (we have simplified the output for readability):

```
(gdb) backtrace
#0 ext4_mb_new_blocks at fs/ext4/mballoc.c:5490
#1 0xffffffff812ce2f3 in ext4_ext_map_blocks ... at
    ↪ fs/ext4/extents.c:4246
#2 0xffffffff812e318e in ext4_map_blocks ... at
    ↪ fs/ext4/inode.c:638
#3 0xffffffff812e70f0 in mpage_map_one_extent ... at
    ↪ fs/ext4/inode.c:2395
#4 mpage_map_and_submit_extent ... at fs/ext4/inode.c:2448
#5 ext4_writepages at fs/ext4/inode.c:2800
#6 0xffffffff81193794 in do_writepages ... at
    ↪ mm/page-writeback.c:2352
#7 0xffffffff8124d3a7 in __writeback_single_inode ... at
    ↪ fs/fs-writeback.c:1467
#8 0xffffffff8124daab in writeback_sb_inodes ... at
    ↪ fs/fs-writeback.c:1732
#9 0xffffffff8124dedb in wb_writeback ... at
    ↪ fs/fs-writeback.c:1905
#10 0xffffffff8124e822 in wb_do_writeback ... at
    ↪ fs/fs-writeback.c:2050
#11 wb_workfn (work=...) at fs/fs-writeback.c:2091
#12 0xffffffff810887b4 in process_one_work (worker=... ,
    ↪ work=...) at kernel/workqueue.c:2276
#13 0xffffffff81088d4d in worker_thread (__worker=...) at
    ↪ kernel/workqueue.c:2422
#14 0xffffffff8108dc8b in kthread (_create=...) at
    ↪ kernel/kthread.c:313
#15 0xffffffff81001a12 in ret_from_fork () at
    ↪ arch/x86/entry/entry_64.S:294
(gdb) frame 13
(gdb) print worker->desc
\$10 = "flush-8:0", '\000' <repeats 14 times>
```

We also leverage strace to examine how bash's redirect operator works:

```
$ strace -r bash -c "echo AAAA > file"
```

From the above command, we get the following part of its output:

```
openat(AT_FDCWD, "file", O_WRONLY|O_CREAT|O_TRUNC, 0666) = 3
fcntl(1, F_GETFD) = 0
```

```
fcntl(1, F_DUPFD, 10) = 10
fcntl(1, F_GETFD) = 0
fcntl(10, F_SETFD, FD_CLOEXEC) = 0
dup2(3, 1) = 1
close(3) = 0
newfstatat(1, "", {st_mode=S_IFREG|0644, st_size=0, ...},
    ↪ AT_EMPTY_PATH) = 0
write(1, "AAAA\n", 5) = 5
```

So we see that bash's redirection seems to be implemented by assigning the file descriptor of the file to be written, to the standard output's fd, using the command dup2. This way, we assume that writing to the open file is not done via some syscall to the VFS, but depends more on the tty driver implementation.

The driver probably maintains a buffer for the output of the command we wish to redirect, which buffer is subject to flushing at some point, based on the working of memory management. This is indicated by functions like `wb_do_writeback` in `backtrace`'s output. But the important thing is that this is done from a kernel worker (`worker_thread` in `backtrace`'s output) which undertakes the data flushing, as it can be seen directly from the description of the worker.

In conclusion, we conclude that testing our ext4 implementation through Bash's redirection, seems to completely bypass the DAX feature logic (since memory is always included in the write path). Also, the node that finally performs the allocation is not that of Bash's instance performing the redirection, but that of the kernel worker to which the specific job is assigned. If there is no kernel worker readily available for the desired node, eventually the file will end up somewhere else than intended.

10.3 More fine-grained DAX support

The DAX implemetation for the ext4 filesystem is a feature that has morphed in multiple ways between different versions of the Linux kernel. Originally, for example in version 5.1, the filesystem supported enabling DAX for all of its files or none of them. Later, as seen in versions like 5.10, the DAX feature could be set on a per-file basis, but to actually enforce the setting on a file it had to stop being referenced by all processes, and later either perform a drop-caches operation (refresh the filesystem data in DRAM), remount the filesystem, or reboot the system [8]. In more recent versions, such as 5.13, for the setting to be enforced it is sufficient that the file is not open by any process [9].

So, we see that the trend is for the management of the DAX attribute to become more and more fine-grained. Ideally, we would like the setting to be applied immediately, that is without any strict condition on whether the file is open by

some process or not. But how this can be implemented is not immediately obvious, as it requires more universal knowledge of ext4's design, especially regarding synchronization.

We can imagine that if the DAX flag is set from active to inactive for some file, it is enough to just allow pages to be stored in the page cache. For the other way around, i.e. activating the DAX feature for a file, the situation is more complicated because a proper flush of the relevant page cache data is required before we start accessing it directly, even when the file is probably used by multiple processes. This would have an immediate performance penalty for the processes that have the file open, especially if our implementation tries to flush all the file data in the page cache on-site instead of on demand, i.e. flush a few pages at a time when some application requests relevant access. This design, of course, presents various problems. The situation becomes drastically more difficult if we also take into account compatibility with the `mmap` interface.

The first case we described, i.e. the transition from DAX active to inactive, was studied for the purposes of this thesis. The relevant additions made are shown in lines `linux-5.13/fs/ext4/file.c:103-106` and `linux-5.13/fs/ext4/file.c:682-687` of our source code. The imposed condition is simple: it is sufficient that the file currently has the DAX attribute activated (inode flag `S_DAX`) and at the same time someone has requested to change this attribute (ext4 inode flag `EXT4_INODE _DAX`). For the sake of completeness, we assume that we would normally also need to check that the file is not already in use by some `mmap` function. For the transition itself, it is sufficient to update the `S_DAX` flag and then the address space operations via function `ext4_set_aops`.

Even in this simple case, attention is needed for synchronization issues. For example, in function `ext4_file_write_iter` it is necessary to proceed while holding the inode lock, since it is not already claimed as in the case of `ext4_dax_read_iter`. We tested our modifications through quick and simple methods, but they appear to be stable (which is not the case unless we update the address space operations), and correct in the sense that the transition is successful even while the file is open.

It would certainly be more interesting to attempt the implementation of the other direction (enabling the DAX feature), but this was far beyond the scope of this thesis, in regards to time contraints. If we had an implementation for both directions, it would allow the substantial modeling of a system in which both the DAX feature and the page cache would be utilized for more efficient management of the aggregate bandwidth of volatile and non-volatile devices, when there are files with high repeatability in their accesses.

Bibliography

[1] *2nd Gen Intel®Xeon®Scalable Processors: Specification Update April 2023*. URL: https://cdrdv2-public.intel.com/338848/338848_2nd%20Gen%20Intel%C2%AE%20Xeon%C2%AE%20Scalable%20Processors%20Specification%20Update_Rev027US.pdf (visited on 07/10/2023).

[2] *2nd Generation Intel®Xeon®Processor Scalable Family based on Cascade Lake product: Reference for hardware events that can be monitored for the CPU(s)*. URL: https://perfmon-events.intel.com/cascadelake_server.html.

[3] *3D XPoint Wikipedia article*. URL: https://en.wikipedia.org/wiki/3D_XPoint.

[4] *3D XPoint™: A Breakthrough in Non-Volatile Memory Technology*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[5] Brian Beeler. *Intel Optane DC Persistent Memory Module (PMM)*. URL: https://www.storagereview.com/news/intel-optane-dc-persistent-memory-module-pmm.

[6] *Bigalloc Documentation*. URL: https://ext4.wiki.kernel.org/index.php/Bigalloc.

[7] *CPU FLame Graphs*. URL: https://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html.

[8] *Dax documentation before v5.13*. URL: https://github.com/torvalds/linux/blob/v5.10/Documentation/filesystems/dax.txt.

[9] *Dax documentation from v5.13 and afterwards*. URL: https://github.com/torvalds/linux/blob/v5.13/Documentation/filesystems/dax.txt.

[10] *Device Mapper Administrator's Guide*. URL: https://docs.kernel.org/admin-guide/device-mapper/.

[11] *FIO documentation*. URL: https://fio.readthedocs.io/en/latest/fio_doc.html.

[12] *General Information on the Device Mapper*. URL: https://en.wikipedia.org/wiki/Device%5C_mapper.

[13] Jungwook Han et al. "Is Data Migration Evil in the NVM File System?" In: *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems Companion (ACSOS-C)*. IEEE. 2021, pp. 26–31.

[14] *Hawkeye: GitHub repository*. URL: https://github.com/apanwariisc/x86-MMU-Profiler.

[15] *IPMCTL Documentation: Creation of memory allocation goal*. URL: https://docs.pmem.io/ipmctl-user-guide/provisioning/create-memory-allocation-goal.

[16] Joseph Izraelevitz et al. "Basic performance measurements of the intel optane DC persistent memory module". In: *arXiv preprint arXiv:1903.05714* (2019).

[17] Wenqing Jia, Dejun Jiang, and Jin Xiong. "NapFS: A High-Performance NUMA-Aware PM File System". In: *2022 IEEE 40th International Conference on Computer Design (ICCD)*. 2022, pp. 593–601. DOI: 10.1109/ICCD56317.2022.00093.

[18] Rohan Kadekodi et al. "WineFS: a hugepage-aware file system for persistent memory that ages gracefully". In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 2021, pp. 804–818.

[19] Saurabh Kadekodi, Vaishnavh Nagarajan, and Gregory R Ganger. "Geriatrix: Aging what you see and what you {don't} see. A file system aging approach for modern storage systems". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 2018, pp. 691–704.

[20] Sagar Karandikar et al. "FirePerf: FPGA-Accelerated Full-System Hardware/Software Performance Profiling and Co-Design". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 715–731. ISBN: 9781450371025. DOI: 10.1145/3373376.3378455. URL: https://doi.org/10.1145/3373376.3378455.

[21] *Non-Uniform Memory Access Architecture Wikipedia article*. URL: https://en.wikipedia.org/wiki/Non-uniform_memory_access.

[22] *NOVA extension for supporting multiple NUMA nodes*. URL: https://github.com/utsaslab/WineFS/tree/main/Linux-5.1/fs/nnova.

[23] *Open Virtual Machine Firmware*. URL: https://wiki.ubuntu.com/UEFI/OVMF.

[24] Ashish Panwar, Sorav Bansal, and K Gopinath. "Hawkeye: Efficient fine-grained os support for huge pages". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 347–360.

[25] *Persistent Memory Development Kit*. URL: https://pmem.io/pmdk/.

[26] *proc(5) linux manual page*. URL: https://man7.org/linux/man-pages/man5/proc.5.html.

[27] *The Linux Function Tracer*. URL: https://www.kernel.org/doc/html/v5.0/trace/ftrace.html.

[28] Erez Zadok Vasily Tarasov and Spencer Shepler. *Filebench: A Flexible Framework for File System Benchmarking*. URL: https://www.usenix.org/system/files/login/articles/login_spring16_02_tarasov.pdf.

[29] Ying Wang, Dejun Jiang, and Jin Xiong. "Numa-aware thread migration for high performance nvmm file systems". In: *Proceedings of the 36th International Conference on Massive Storage Systems and Technology*. 2020.

[30] Jian Xu and Steven Swanson. "{NOVA}: A log-structured file system for hybrid {Volatile/Non-volatile} main memories". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. 2016, pp. 323–338.

[31] Jian Yang et al. "An empirical guide to the behavior and use of scalable persistent memory". In: *18th USENIX Conference on File and Storage Technologies (FAST 20)*. 2020, pp. 169–182.