PUCRS

FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

GIANLUCCA OLIVEIRA PUGLIA

EXPLORING ATOMICITY ON MEMORY MAPPED FILES BASED ON NON-VOLATILE MEMORY FILE SYSTEMS

Porto Alegre
2017

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
FACULTY OF INFORMATICS
GRADUATE PROGRAM IN COMPUTER SCIENCE

EXPLORING ATOMICITY ON MEMORY MAPPED FILES BASED ON NON-VOLATILE MEMORY FILE SYSTEMS

GIANLUCCA O. PUGLIA

Thesis presented as partial requirement for obtaining the degree of Master in Computer Science at Pontifical Catholic University of Rio Grande do Sul.

Advisor: Prof. Avelino Francisco Zorzo

Porto Alegre
2017

Ficha Catalográfica

P978e Puglia, Gianlucca Oliveira

Exploring atomicity on memory mapped files based on non-volatile memory file systems / Gianlucca Oliveira Puglia . – 2017.
104 f.
Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Avelino Francisco Zorzo.

1. Memórias não-voláteis. 2. Sistemas operacionais. 3. Mapeamento sistemático. 4. Sistemas de arquivos. I. Zorzo, Avelino Francisco. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Gianlucca Oliveira Puglia

Exploring atomicity on memory mapped files based on non-volatile memory file systems

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre em Ciência da Computação do Programa de Pós-Graduação em Ciencia da Computação, Faculdade de Informática da Pontifícia Universidade Católica do Rio Grande do Sul.

Aprovado em 21 de março de 2017.

BANCA EXAMINADORA:

Prof. Dr. Tiago Coelho Ferreto (PPGCC/PUCRS)

Prof. Dr. Edson Norberto Cáceres (FACOM/UFMS)

Prof. Dr. Avelino Francisco Zorzo (PPGCC/PUCRS - Orientador)

EXPLORANDO ATOMICIDADE EM ARQUIVOS MAPEADOS EM MEMÓRIA BASEADOS EM SISTEMAS DE ARQUIVOS PARA MEMÓRIAS NÃO-VOLÁTEIS

RESUMO

As tecnologias de memórias não-voláteis são uma grande promessa na área de arquitetura de computadores e é esperado que sejam poderosas ferramentas para solucionar os problemas referentes a manipulação eficiente de dados dos dias de hoje. Estas tecnologias provêm alta performance e acesso em granularidade de bytes com a distinta vantagem de serem persistentes. Porém, afim de explorar estas tecnologias em todo seu potencial, os sistemas e arquiteturas de hoje precisam buscar meios de se adaptar a esta nova forma de acessar dados e de superar os desafios que vêm com ela.

Trabalhos existentes na área já propõem métodos para adaptar as arquiteturas existentes para o uso de NVM bem como formas inovadoras de empregar estas memórias em futuras aplicações. No entanto, o suporte dos sistemas operacionais a estas soluções, ainda que existente, ainda é muito limitado. Neste trabalho, nós apresentamos duas variações da chamada de sistema *msync*, modeladas para explorar as características das tecnologias de NVM e garantir consistência para os dados dos usuários. Ambas são soluções simples que permitem aos usuários definirem checkpoints de seus arquivos usando a sintaxe comum de sistemas de arquivos. Nós implementamos e testamos estes métodos sobre o sistema operacional Linux utilizando como base um sistema de arquivo nativamente voltado a NVM. Nossos resultados mostram que estes mecanismos são capazes de garantir a integridade dos arquivos mesmo na presença de falhas no sistema enquanto mantém uma performance razoável.

Palavras-Chave: Memórias Não-Voláteis, Sistemas Operacionais, Mapeamento Sistemático, Sistemas de Arquivos .

EXPLORING ATOMICITY ON MEMORY MAPPED FILES BASED ON NON-VOLATILE MEMORY FILE SYSTEMS

ABSTRACT

Upcoming non-volatile memory technologies are a big promise in computer architecture and are expected to be powerful tools to address today's issues regarding efficient data manipulation. They provide high performance and byte granularity while also having the distinct advantage of being persistent. However in order to explore these technologies to their full potential, existing systems and architecture must adapt to this new way of working with data and workaround the challenges that come with it.

Existing work in the area already proposes methods to adapt existing architecture to NVM as well as innovative ways to employ these memories in future applications. However operating system support to such NVM-enabled solutions, although existent, still very limited. In this work, we present two variations of the existing *mmap* system call, designed to both explore NVM characteristics and provide user data consistency. Both are very simple solutions that allow users to control the persistence and define checkpoints to their files while using the common mapped file syntax. We have implemented and tested these methods over Linux using a NVM file system as our base. Our results show that these mechanisms can ensure file integrity in the presence of system failures while also providing a reasonable performance.

Keywords: Non-Volatile Memory (NVM), Operating Systems (OS), Systematic Mapping Study (SMS), File Systems.

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1. INTRODUCTION

The increasing disparity between processor and memory performances led to the proposal of many methods to mitigate memory and storage bottlenecks [98][5]. Moreover, the research advances in the so called Non-Volatile Memories (NVMs) in recent years suggest that these technologies are quite promising and may eventually replace the entire memory hierarchy, drastically changing computer architecture as we know today. One possibility of such architectural shift based on NVM technology, for example, would be a radical memory-centric architecture [33]. NVMs provide performance speeds comparable to those of today's DRAM (Dynamic Random Access Memory) and, like DRAM, may be accessed randomly with little performance penalties. Unlike DRAM, however, NVMs are persistent, which means they do not lose data on power loss. In summary, NVM technologies combine the advantages of both DRAM and Hard Disk Drives (HDDs).

These NVMs, of course, present many characteristics that make them substantially different from HDDs. Therefore, working with data storage in NVM may take the advantage of using different approaches and methods that systems designed to work with HDDs do not support. Moreover, since the advent of NAND flash memories, the use of NVM as a single layer of memory, merging today's concepts of main memory and back storage, has been proposed [89] [90], aiming to replace the whole memory hierarchy as we know. Such change in the computer architecture will certainly represent a huge shift on software development as well, since most applications and operating systems are designed to store persistent data in the form of files in a secondary memory and to swapp this data between layers of faster but volatile memories.

Even though all systems running in such an architecture would inevitably benefit from migrating from disk to NVM, one of the first places one might look at, when considering this hardware improvement, would be the file system. The file system is responsible for organizing data in the storage device (in the form of files) and retrieving this data whenever the system needs it. File systems are usually tailored for a specific storage device or for a specific purpose. For instance, an HDD file system, like Ext4, usually tries to maintain the data blocks belonging to a file contiguously or at least physically close to each other for performance reasons. A file system designed for flash memory devices, like JFFS2, might avoid rewriting data in the same blocks repeatedly, since flash memory blocks have limited endurance. The list could go on, but we can conclude that whenever the storage hardware changes, the way data is stored and accessed must be reviewed. In all these cases, file systems should be adapted to this new conditions in order to reduce complexity and to achieve the best possible performance.

Concerning this adaptation for NVM, in the 2013 Linux Foundation Collaboration Summit [22], the session "Preparing Linux for nonvolatile memory devices" proposed a three-step approach. In the first step, file systems will access NVM devices using traditional block drivers. This step does not explore the disruptive potential of NVM but is a fast way of making it accessible. In the second step, existing file systems will be adapted to access NVM directly, in a more efficient way. This step ensures that file systems are designed for NVM, but keep the traditional file system abstraction for

compatibility purposes. The final step is the creation of byte-level I/O APIs for new applications to use. This step will explore interfaces beyond the file system abstraction that may be enabled by NVM. It has the most disruptive potential, but may break backward compatibility.

With this scenario in mind, we propose our contribution by exploring the needs and limitations of current NVM file systems. First we survey the current state of the area of NVM file systems, identifying challenges and issues as well as trends and potential research topics. Next we explore and discuss these topics, pinpointing fundamental questions regarding NVM adoption and prospecting future developments in the area. Finally, we propose our own solution, which focus on providing future applications with basic atomicity constraints on the OS level while also exploring the impact of using volatile RAM as write buffer for file updates.

We present two new methods of mapping files to process memory based on mechanisms already consolidated by existing NVM file systems implementations. These methods seek to allow applications to access persistent data efficiently by directly manipulating it in NVM while also ensuring file integrity and atomicity. We implement and evaluate our solutions on a range of different scenarios in order to explore its behavior as well as its limitations.

This work is organized as follows:

- Chapter 2 provides some basic background on NVM file systems and its related concepts.
- Chapter 3 details the process used in this work for selecting, analyzing and classifying studies on the NVM file system topic.
- Chapter 4 presents the results of our survey process and discusses our findings.
- Chapter 5 shows an overview of the state-of-the-art of NVM file systems which is also the base for our proposal.
- Chapter 6 describes our proposed solutions and its implementation details.
- Chapter 7 presents and discusses the results we have obtained by submitting our implementation to file system benchmarks and failure tests.
- Chapter 8 concludes this document, summarizing the obtained results and the expectations for future works.

2. BASIC CONCEPTS

This chapter is dedicated to the presentation of fundamental concepts behind this study. These concepts are all intimately connected to each other and are also the base knowledge that supports the idea of NVM file system. Further information regarding the context in which our work is developed is presented in the coming chapters.

2.1 Non-volatile Memories

As the name might suggest, Non-Volatile Memory (NVM) technologies are storage devices that provide access latency close to DRAM devices while also offering the persistence of traditional HDDs. NVM is a relatively new and emerging technology that is expected to reduce the impact of the storage bottleneck on the current architecture. The ever growing demand for data of today's applications poses a challenge for architectures and technology as they must find the means to supply large amounts of data with high throughput. It is in this scenario that NVM technologies are presented as a solution, able to provide a large persistent area with DRAM-like performance.

Besides its performance, NVM technologies are also designed to be more dense than volatile-memories, which means that NVM devices may scale to much larger spaces than today's DRAM devices. Furthermore, NVM is byte-addressable and may be accessed by processors through regular memory-bus interface. With these characteristics, NVM may be used as either main memory or as storage. This enables architectures with large arrays of memory that may also be used for persistent data, however without much of the overhead and complexity of I/O layers like block devices and schedulers. It brings persistent data (like files) much closer to the processor and to applications, drastically reducing the bureaucracy presented by file system operations and asynchronous I/O transfers .

With this design comes a few challenges though. Current architecture, processors and software are not yet suited to properly work with large amounts of persistent memory in the memory-bus [33][4]. There are many well-known issues with this approach, for example, problems related to addressing huge memory spaces, protecting persistent data in NVM from corruption and securing sensitive data in NVM, to name a few. Researchers are already looking into these matters [100] [56], but there are still questions with no answers and much potential to be explored.

NVM technologies also have some limitations of their own, like limited endurance and asymmetry in performance and energy consumption. Writing in NVM cells is significantly more expensive in terms of time and energy than reading from them. Different solutions have been proposed to mitigate the impact of these writes or reduce the number of writes that actually reach the NVM device [44][101]. This characteristic is also one of the reasons behind the designs that employ NVM along side DRAM creating a hybrid memory layer. Among other things, this approach

allows regular volatile memory to be used in cases where using NVM would be unfeasible (like temporary data or frequently updated data and metadata).

In this paper we explore the most popular and mature of these memory technologies, focusing on the general key aspects shared by NVM. There are other alternatives that have been under development [65], which we do not describe. The technologies we cover are:

- Ferroelectric RAM: in FeRAM, ferroelectric material (the most common being lead zirconate titanite or PZC) is used to create a ferroelectric capacitor (FeCap) to hold data persistently on FeRAM cells [65]. The state of the cell is identified by the cell's electric polarity and may be written or read through the application of electric fields. FeRAM presents no power leakage and therefore its cells do not require to be refreshed, which significantly reduces these devices overall energy consumption. FeRAM also presents latency very close to that of DRAM and better endurance when compared to other NVMs, *e.g.* Flash. The main disadvantage of FeRAM is in its low capacity, which is somewhat similar to that of DRAM. This poses a challenge to the adoption of such memories as long term storage.

- Magnetic RAM: Also known as Magnetoresistive RAM, Magnetic RAM (MRAM) is basically composed of two magnetic tunnels whose polarity may be alternated through the application of a magnetic field. Conventional MRAM (also called "toggle-mode" MRAM) uses a current induced magnetic field to switch the Magnetic Tunnel Junction (MTJ a structured basically composed of two magnetic tunnels whose polarity may be alternated through the application of a magnetic field) magnetization. The amplitude of the magnetic field must increase as the size of MTJ scales, which compromises MRAM scalability. Like FeRAM, MRAM presents high endurance (over 10^{15} writes) and extremely low latencies (faster than DRAM) [130]. Also, the energy necessary to read and write from MRAM cells is higher than on DRAM cells, hence it is usually considered that MRAM is more energy efficient than DRAM due to its lack of need for cell refresh. However MRAM suffers from a severe issue of density which prevents it from scaling to storage levels. For that reason, much research was invested into exploring new memory architectures to make MRAM usage more feasible.

- Spin-Torque Transfer RAM: Spin-Torque Transfer RAM (STT-RAM) is a variation of MRAM, designed to address the scalability issues of its predecessor. The main difference between these two technologies is in the cell write process: in STT-RAM, a spin-polarized current, instead of a magnetic field, is used to set bits, which makes the cell structure much simpler and smaller. Similar to MRAM, both the efficiency and endurance in STT-RAM are excellent, being able to achieve latency lower than DRAM [65] and number of writes superior to Flash. The main challenge in adopting STT-RAM at large scale is due to its low density, even though some authors agree that the technology has a high chance to replace existing technologies such as DRAM and NOR Flash [65].

- Resistive RAM: The basic concept behind Resistive RAM (RRAM) technology is similar to that of MRAM in that the electric resistance of components are modified by external operations to change the state of the memory cells. The most typical method to do that is applying different voltage levels to change the cell resistivity. In general, RRAM is known to be quite efficient, both in terms of access latency and energy. One of the main advantages of these technologies however, is their scalability that, supposedly, may easily surpass that of DRAM. The main drawback of RRAM devices is their limited endurance. Reportedly, resistive technologies such as Memristor can get around 10^7 [130] writes of lifetime, which may limit the usage of the technology (as main memory, for example).

- Phase-Change RAM (PCRAM): Phase-Change Random Access Memory (also called PCRAM, PRAM or PCM) is currently the most mature of the new memory technologies under research. It relies on phase-change materials that exist in two different phases with distinct properties: an amorphous phase, with high electrical resistivity, and a crystalline phase, with low electrical resistivity [104]. PCRAM scales well and presents endurance comparable to that of NAND Flash, which makes it a strong candidate for future high-speed storage devices. This technology is slower than DRAM (between 5 and 15 times slower) and has a considerable disparity in energy consumption due to its RESET operation dissipating a larger amount of energy than other operations [104] [8].

- Flash Memory: Flash memory is the most popular and wide-spread technology on this list. The original Flash memory structure was designed after traditional Electric Erasable Programmable Read-Only Memory (EEPROM) to be able to perform erase operations over separate blocks, instead of over the entire device. Flash memory is mainly divided into NOR and NAND Flash. While NOR Flash is faster and may be written (but not erased) at byte granularity, NAND presents superior density [14] and is significantly more durable. In general, Flash memory is known for being several times slower than emerging byte-addressable NVM technologies and usually employed as an I/O device, replacing magnetic disks. Despite that, Flash does share a few key characteristics with upcoming NVM technologies, such as limited endurance, density, energy constraints, different speeds for read and write and persistence. Since its introduction, Flash memories have been extensively studied and a variety of mechanisms to both cope with and explore its characteristics have been proposed [26][49][55]. This research notably influenced current under development NVM technologies. Hence, we argue that, although Flash memory may present significantly distinct characteristics when compared to upcoming byte-addressable NVM, knowledge in many aspects and topics regarding Flash (such as wear-leveling, garbage collection, log-structured file systems and address translation layer, to name a few) may be useful to understand and guide research on emerging persistent memory technologies. Furthermore, Flash memory is still currently extremely popular and it does not seem likely that SSDs (Solid State Drives) are going to get obsolete anytime soon.

Upcoming NVM technologies have much in common individually (such as low energy leakage, fast access, efficient random access and lifetime limitations), but each one of them present some kind of weakness: some have issues with endurance, some have lower performance, some do not scale well. Table 2.1 summarizes the main characteristics of these memory technologies. Additionally they are at different stages of development, some being studied in laboratories only, while some are already being commercialized. All of these memories have a real potential to replace current predominant technologies at some level of the memory hierarchy (such as HDDs for storage, DRAM for main memory and SRAM for processor caches) and they all (with the exception of Flash, perhaps) represent a huge shift in how persistent data is managed on today's systems. Therefore researchers have been thoroughly exploring the potential of these technologies and proposing solutions that may either overlap or complement each other. That being said, in this work, we do not focus in any particular NVM technology, even though we emphasize innovations and studies on byte-addressable NVM as our main interest.

Table 2.1: Characteristics of the NVM technologies discussed in this work.

	DRAM	FeRAM	MRAM	STT-RAM	RRAM	PCRAM	Flash
Density per Chip	8 - 16 Gb	128 Mb	16 - 32 Mb	2 - 16Mb	64Kb	1 Gb	256 - 512Gb
Endurance	10^{15}	10^{15}	10^{15}	10^{15}	10^{5}	10^{7}	10^{4}
Read Latency	10 - 60ns	75ns	5 - 10ns	5 - 10ns	10ns	50ns	25µs
Write Latency	10 - 60ns	50ns	12ns	12ns	10ns	40 - 150ns	200µs
Energy per Write	2 pJ	2 pJ	120 pJ	0.02 pJ	2 pJ	100 pJ	100 - 1000 mJ

2.2 File Systems

One of the simplest methods to provide access to NVM to applications is by simply mounting a file system over it. Using special block drivers, it is possible to build traditional disk file systems, like ReiserFS, XFS or the EXT family, over a memory range. Metadata and namespace management is made by the file system while the block driver is responsible for the actual writes to the physical memory. However, since these file systems were designed for much slower devices with very different characteristics, they usually are not the best fit for NVM management. With this in mind, a handful of alternative file systems, designed specifically for NVM, were proposed, designed and implemented [32][21][126][128]. NVM file systems usually take in account issues such as minimizing processor overhead, avoiding unnecessary data copies, tailoring metadata to NVM characteristics and ensuring both data protection and consistency.

BPFS [21] and PMFS [32] are two early and well known examples of NVM improved file systems, designed to provide efficient access to NVM with POSIX interface. Both systems are designed for memory-bus attached NVM storage and attack common NVM-related issues such as efficient consistency mechanisms, consistency with volatile processor cache and NVM optimized structures. Among other things, BPFS proposes the epoch barrier mechanism, to reinforce ordering

and maintain consistency when writing to NVM while also avoiding cache flushes, and the short-circuit shadow paging which is a fine-grained NVM-friendly redesign of the traditional shadow paging. PMFS, on the other hand, employs fine-grained journaling to ensure atomicity on metadata updates while adopting the copy-on-write technique to ensure atomicity on file data updates. PMFS also provides memory protection over the file system pages by marking them as read-only and allowing them to be updated by kernel code only when necessary, during small time windows, by manipulating the processor's write protect control register.

A more recent example of NVM designed file system is NOVA [128]. Besides improving the efficiency of file system structure and operations based on NVM characteristics NOVA also seeks to provide support for NVM scalability. It minimizes the impacts of locking in the file system by keeping per-CPU metadata and enforcing their autonomy. Like BPFS and PMFS, NOVA keeps some of its structures in DRAM for performance reasons while also ensuring the integrity of metadata stored in NVM. NOVA is also log-structured which is a common structure of file systems for persistent memory due to their affinity with these technologies.

In addition to its simplicity and straightforwardness, the adoption of file systems is also important to maintain a consistent interface with legacy software and to make data sharing easy. Much of the interaction of today's applications with persistent data is highly coupled with the specification of file system operations. Although NVM file systems may eventually evolve beyond the traditional norms of POSIX, it is important to keep compatibility with legacy code in mind when redesigning and optimizing the access interface for NVM. In this scenario, even with the emergence of alternative more memory friendly persistence models, like persistent heaps, file systems are still essential for working with NVM.

2.3 File Mappings

Mapping files to memory regions is a common practice in modern software, being a flexible and efficient alternative to traditional read/write file system operations. First the file is mapped to a region of main memory. Accesses (reads or writes) to this region generate page faults, so the operating system may perform the necessary I/O and addressing to allow the file's data to be accessed in a memory-like fashion through virtual addresses. Unless specified otherwise, the file is mapped on demand: every access to a new page generates a page fault for the kernel to adjust the process page table. The operating system is also responsible for writing back the file's in-memory dirty pages to the file system, thus not requiring any additional system calls from the user.

Naturally, in traditional systems where storage is based on HDDs and SSDs, much of this work is made asynchronously, as swapping pages to and from the backing store are extremely slow operations. For instance, mapping a file region using the *mmap* function creates the appropriate operating system data structures, such as virtual memory areas (VMA) and page table entries, in preparation for the incoming data. Before the pages of the mapped file can be accessed by the

process, they must be loaded into the page cache from the file system. Once the pages are in memory, the operating system modifies the process page table to point to the new copied pages in page cache. When pages belonging to a mapped file become dirty, they become eligible to be asynchronously copied back to their original file. Handling page table entries, dirty pages, I/O and writebacks are all processes managed by the operating system, involving a minimum amount of interaction from the user. In most cases this is very convenient as reducing the amount of calls to the operating system (and consequentially the amount of context switches) may significantly improve performance. On the other hand user control and guarantees over writes and ordering are reduced and the additional overhead of manipulating (among other things) page table structures may be considered drawbacks in some situations.

Regarding NVM storage, the primary issue with the described mapping process is that it involves unnecessary I/O and processing that does not quite fit with a NVM enabled architecture. Since NVM may be addressed in byte granularity by the processor and accessed with almost DRAM speed, copying and swapping data blocks from NVM to main memory seems redundant. Furthermore, the expensive and bureaucratic I/O operations over which *mmap* is based, do not match the much less complex and transparent access structure allowed by NVM. Based on these arguments, some file systems offer the eXecute-In-Place (XIP) functionality. With XIP, entire layers of storage subsystem, like I/O schedulers and page cache management, may be bypassed when accessed data in memory-bus attached NVM. These are unnecessary layers that are not designed for NVM-based storage and which would otherwise only slow down the file system operations. The result is a much cleaner and efficient method to interface with NVM file systems and NVM storage in general.

Furthermore, adopting the XIP mechanism also allows applications to map files directly in their address space, in a true zero-copy fashion. A simple implementation of such XIP-enabled mappings is presented by PMFS. In PMFS, during page faults on a mapped area, instead of copying the faulting block from the file into the page cache, the file system instead adds to the process page table, an entry pointing to the block's physical address. This means that writes are made directly to the file and no paging is necessary. The mapped block is represented by a page frame number and is managed by PMFS instead of the memory management subsystem.

Given its advantages and particularly its flexibility, the direct mapping mechanism is used as basis for more robust and complex data store models. Tools such as persistent heaps [43][20] and persistent regions [118] have a facilitator in the direct mapping functionality. They see direct mappings as a simple mean to have direct access to a NVM region with minimum interference from the operating system. The persistent regions presented by Mnemosyne, for instance, are actually structures backed by a mapped file and managed by the Mnemosyne library. Frameworks like this aim to provide more programming-friendly APIs (as an alternative to files semantics) as well as more complex functionalities and guarantees (such as data integrity and atomicity), while delegating the low-level management to the file system. It seems likely that, as NVM devices become more mature and accessible, these kinds of tools may grow in popularity which would consequentially generate demand for file system resilience and efficiency as well.

3. SYSTEMATIC MAPPING STUDY

The first part of this work is dedicated to give an overview of the current state of research in the area of NVM file systems. This overview will help us to develop the technical base and motivation for this work and to situate it in the picture of current NVM research. To this end, we adopt the model of Systematic Mapping Study (SMS), in order to build a framework of research which helps both to reduce researcher bias and to allow replication in the future.

This chapter presents the protocol used in the systematic mapping study and how the research was conducted. The method used by the SMS was based on processes used by other similar studies conducted in both Computer Science and Software Engineering [98] [5] [58]. The first step of this method is to define a protocol that presents some details of how the research was conducted and how the selection of the analyzed studies was made using paper search engines and manual searches. The idea of specifying a protocol may help future works and secondary researches in the field, by providing a well-structured method of retrieving the studies used by this systematic mapping. It also provides the context in which the research was conducted and helps to visualize the goals and the scope of the systematic mapping, and how its results may be of use.

For the sake of completeness, during the execution of this survey, we broaden the meaning of the term Non-volatile Memory to also incorporate technologies such as NAND Flash that are not byte-addressable. Flash memory has much in common with currently under development byte-addressable NVM, and much of the research dedicated to these upcoming technologies has its roots in studies performed over flash. Therefore we do not distinguish byte-addressable NVM from Flash in this SMS although we do focus on the issues common to both technologies.

3.1 Defining Scope

The goal of this survey is to map the state of the art of NVM file systems based on studies conducted in the field. This should help to visualize the problems, challenges and main goals when writing a file system designed for NVM technologies. It also helps to identify the trends of the field and what currently seems to be the future of NVM usage and application, and how it may impact on the overall computer architecture.

3.2 Establishing Research Questions

The use of research questions to guide an academic research is a common approach. These questions help researchers to define the scope of the research, the premises on which the research will be based on and what kind of data, arguments and experiments would represent a satisfactory answer

(results of the research). The research questions also help to identify what kind of contribution the research work will represent to the academic community knowledge.

In a systematic mapping study, the research questions are also used as the basis for the search string, that will be used to query academic databases for papers related to the study's subject. Therefore, establishing a research question is an important part of the systematic mapping protocol. This mapping was based on the following 4 questions:

- **RQ1:** What are the differences between disk-based and NVM file systems?
- **RQ2:** What are the challenges and problems addressed by NVM file systems?
- **RQ3:** What techniques and methods have been proposed to improve NVM file systems?
- **RQ4:** What is the impact of new file system models on the overall architecture?

3.3 Inclusion and Exclusion Criteria

To filter the papers used by the systematic mapping study, a set of inclusion and exclusion criteria is usually applied during the research. This method helps to ensure that only relevant papers will be selected and analyzed.

The inclusion criteria used are the following: (1) studies that provide a substantial comparison between a NVM file system and another file system (designed for NVM or not) (2) studies that propose new NVM file systems or new models of NVM usage, (3) studies that propose improvements over general purpose file systems to work with NVM, and (4) studies that discuss or criticize existing NVM file systems and NVM technologies.

The exclusion criteria are the following: (1) studies not written in English, (2) studies that only mention the subject, but are not focused on it, (3) in case of duplicated or similar studies, only the most recent one was considered, (4) studies that do not mention NVM file systems or NVM technologies in its title and abstract, and (5) studies that only focus on NVM hardware aspects and impacts.

3.4 Research Strategy and Search String

This section describes in detail how the research was conducted. In order to search academic databases for relevant studies, the key terms of the previously defined research questions were extracted and used to create a well-formed search string. The search string was then applied to

selected databases' search engines in order to retrieve papers containing the main specified terms. The set of databases accessed by this systematic mapping study is composed of IEEExplore Digital Library, ACM Digital Library, Springer Link and EI Compendex.

The search string was built by extracting the meaningful terms of the established research questions and organizing them into three groups: population, intervention, and outcome. This is a common method used in medical research, but have been applied in software engineering studies as well. The groups are organized as:

- **Population:** Non-volatile memory. **Synonyms:** NVM, persistent memory, storage class memory, byte-addressable memory.

- **Intervention:** File system. **Synonyms:** filesystem, file-system.

- **Outcome:** Problems and techniques. **Synonyms:** challenges, approaches, models, methods.

To make the search string as comprehensive as possible, "OR" operators were used to establish the relationship between synonyms and similar terms, and "AND" operators were used to connect population, intervention and outcome terms. The terms were also converted to singular for convenience. The resulting search string is the following:

("non-volatile memory" OR "NVM" OR "persistent memory" OR "storage class memory" OR "byte-addressable memory") AND ("file system" OR "filesystem" OR "file-system") AND ("problem" OR "technique" OR "challenge" OR "approach" OR "model" OR "method")

After executing the search string in the aforementioned search engines, the next step taken was to apply the insertion and exclusion criteria over the retrieved studies. To fit these studies in the proposed criteria, information like publication year, paper title and abstract were read and analyzed. Additionally, similar or redundant studies were discarded by selecting only the most recent one. The output of this process is the set of primary studies that are going to be addressed by this survey.

To further broaden the range of material used by this SMS, relevant studies referenced by the primary studies were also verified and, when appropriate, selected to be used in the systematic mapping as well. The same criteria were applied over these referenced studies.

3.5 Applying the Search String

As explained previously, the research was conducted by querying four search engines (ACM Library, EI Compendex, IEEExplore and SpringerLink) using the search string presented in Section

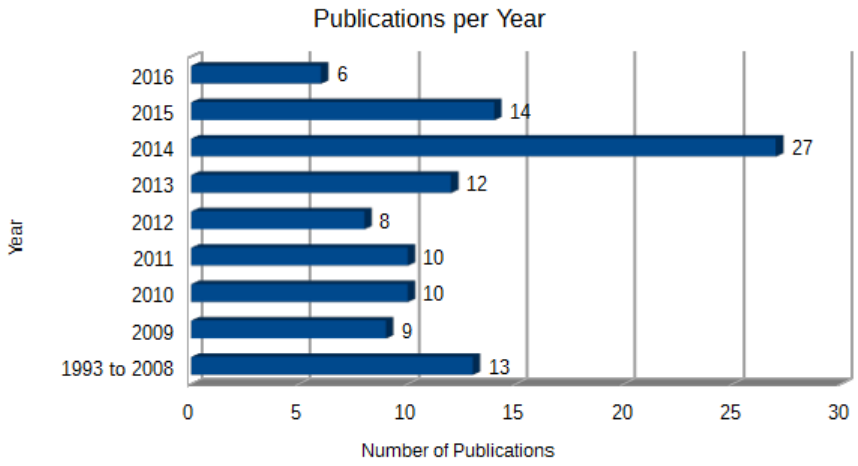

Figure 3.1: Distribution of publications by year. Partial amount: number of publications relative to July of 2016.

Table 3.1: Number of retrieved/selected studies by each search engine.

Engine	Number of Retrieved Studies	Number of Selected Studies
IEEExplore	105	42
ACM Library	105	35
Compendex	64	12
Springer Link	198	20
Total	472	109

3.4. Together, the four engines returned a total of 472 publications (including duplicates) from which 109 were selected based on the inclusion and exclusion criteria. To apply the criteria and select appropriate publications, the information in the title and abstract were used. Table 3.1 shows the distribution of publications retrieved and selected for each search engine.

In this research, no date restrictions were specified. Figure 1 presents the number of papers per year of publication. The figure shows that, though some papers in the NVM area date from as far as 1992, most of the retrieved papers were published from 2008 and 2015. Additionally, some of the most relevant papers (by number of citations) in the field date from before 2008 (like eNVy [27], for example). Therefore it seems that not imposing a date range was the best option in this case. The figure also shows a clear intensification of research in the NVM area in the last 6 years. This is probably due to the growing popularity of NAND Flash Solid-State Disks (SSDs) and the growing maturity and promising specifications of new NVM technologies, especially the Phase-Change RAM (PCRAM) [127] and Spin-Transfer Torque RAM (STT-RAM) [85].

3.6 Collecting and Classifying the Results

This section presents the next step in the SMS, which involves analysis and classification of the retrieved studies shown in Section 3.5. The classification presented here is driven by the

previously established research questions, and is intended to provide a detailed view of the selected studies in order to answer those questions.

Table 3.2: Selected studies categorized by the type of their contributions.

Contribution	Occur.	Studies
File System	26	[26], [37], [7], [70], [61], [89], [114], [21], [129], [1], [40], [90], [126], [32], [132], [92], [52], [28], [119] [39], [51], [29], [96], [111], [12], [69], [108], [128]
Alternative software layer	12	[55], [15], [76], [63], [43], [35], [82], [118], [54], [68], [74], [67], [107]
Techniques	45	[27], [52], [12], [15], [76], [63], [82], [54], [72], [124], [10], [3], [106], [57], [2], [45], [49], [42], [83], [109], [79], [88], [38], [103], [13], [16], [48], [44], [64], [77], [80], [125], [73], [95], [18], [75], [100], [91], [113], [60], [62], [78], [105], [81], [120], [46], [56], [116]
Architecture	28	[89], [90], [27], [70], [129], [1], [132], [28], [119], [51], [29], [96], [111], [10], [42], [103], [16], [44], [91], [120], [131], [9], [11], [30], [59], [19], [112], [101], [46]
Alternative applications of NVM	16	[37], [7], [132], [52], [54], [68], [3], [57], [83], [103], [64], [60], [30], [17], [53], [41]
Aggregated knowledge	13	[127], [85], [9], [17], [86], [97], [102], [36], [93], [94], [4], [34], [66], [71], [31], [123], [110]

Table 3.2 classifies the studies according to the type of proposals and contributions they present. The categories listed in Table 3.2 were derived from the approach patterns identified while analyzing the retrieved studies. These categories reflect the major concerns when working with NVM and the research effort invested by the academic community on different approaches to deal with these concerns. The table also shows that many different applications of NVM technology have been proposed, ranging from non-volatile caches to completely NVM hierarchies. It is also important to notice that the same study may be related to more than one category, meaning that that study makes multiple proposals (that probably complement each other) in different aspects of the NVM usage. The categories of contribution identified and used by this work are the following:

- *File systems:* publications under this category focus their efforts on proposing the use of a file system designed specifically for persistent memories. The file system may be an alternate version of an existing file system adapted to work with persistent memory, or may be a new file system designed to work with NVM.

- *Alternative software layer:* although a file system may be the first piece of software we consider when studying the impact of new NVM technologies over the traditional architecture, there are other software layers and subsystems responsible for managing memory and storage.

These software layers intend, for example, to allow traditional file systems to work with NVM more efficiently. Common examples are object-based storage systems (both as an alternative to file systems or as a complementary software layer), translation layers and block drivers.

- *Techniques:* this category is reserved for those studies that instead of presenting a complete software layer, propose some methods that aim to address specific problems or improve the usage of NVMs. These methods are usually integrated into existing systems (e.g. a file system) for evaluation. Techniques detailed by these studies are usually driven by common concerns related to memory and storage management, like metadata storage improvement, block allocation algorithms or wear leveling.

- *Architecture:* the majority of the studies involving NVM assume the use of persistent memory in two architecture models: (1) where NVM is used as storage, replacing traditional HDDs, and (2) where NVM is used as main memory, replacing partially or completely the DRAM-based main memory. In the scope of this work, it is considered that any publication that presents an architecture different of (1) and (2) is proposing a new architecture or an architecture adaptation and, therefore is categorized under this category. Some examples of these architecture proposals involve the use of NVM to improve metadata management and mixing NVM with volatile memory for performance reasons.

- *Alternative NVM applications:* the most common and straightforward application of emerging NVM technologies is to either use them as a block device to provide high speed storage, or to plug them into the main memory bus, giving the CPU direct access to the NVM and using it as a persistent main memory. Studies of this category explore the application of these same NVM technologies in other levels of the memory hierarchy, like buffers and caches. Differently from the Architecture category, these studies do not focus on NVM as a mean of storage, but as a mechanism to improve performance and reliability of storage and RAM devices, complementing their functionalities. Also, theses studies are usually driven by limitations of current NVM (related to density, cost and durability) or focused on embedded systems. Alternative applications addressed by these publications include persistent processor cache and persistent buffer for disks.

- *Aggregated knowledge:* publications under this category do not focus on proposing methods to improve NVM usage. Instead, these works present aggregated knowledge over the NVM area, exploring characteristics of NVMs and the impact of these technologies in the many aspects of the current computer architecture.

The purpose of the classification presented in Table 3.3 is to identify different applications that researchers believe that could benefit from the advantages of NVMs. Although it seems obvious that any data-intensive application, like databases, search engines and data mining applications can get significant improvements from persistent memory, efficient methods to explore the full potential of NVM under these workloads are still under invetigaton. Furthermore, in their current state, NVM technologies still present limitations in density and resilience that prevents them from being adopted in large scale. Besides, these technologies have a few other characteristics aside from its low latencies that may represent significant improvements in other fields of application. The most foreseeable case is that of embedded and mobile systems, that may benefit from memories that are impact resistant and energy efficient, while also being byte-addressable.

Table 3.3: Selected studies categorized by their area of application.

Application	Studies
General purpose	[89], [90], [27], [127], [85], [61], [114], [21], [129], [126], [32], [132], [92], [28], [119], [39], [51], [111], [12], [69], [55], [15], [63], [43], [35], [82], [118], [54], [68], [67], [72], [124], [10], [3], [83], [109], [38], [16], [48], [44], [100], [60], [131], [53], [86], [97], [102], [36], [4], [66], [116] [71], [31], [123], [108], [56], [110], [107], [128]
Embedded Systems	[26], [70], [40], [96], [2], [64], [77], [81], [30], [101], [93], [94]
Mobile Systems	[29], [106], [2], [45], [38], [13], [77], [80], [125], [18], [62], [105], [59], [112]
Distributed systems/Clusters	[7], [52], [57], [103], [95], [11], [17], [46]
HPC/Scientific Applications	[52], [57], [78], [120], [9], [41], [34], [46]
Data intensive applications/Databases	[27], [37], [42], [9]
Large and Long Running Systems	[91], [113]
Streaming Systems	[88], [59]
Others	[26], [1], [76], [74], [42], [73], [75], [19]

Table 3.3 also shows that most solutions proposed in the selected studies do not focus on a single niche. Instead, they focus on providing solutions to common NVM-related problems or problems related to a specific architecture, and presenting alternative applications of NVM. This does not necessarily mean that these solutions may not benefit specific applications, but that they were not developed having a specific class of problems in mind, or at least none was specified by the authors in their respective papers.

The solutions that are not categorized as general purpose may be divided in basically 3 categories: mobile, embedded and distributed systems. Studies that focus on mobile systems usually explore the benefits of low power consumption and byte-addressability of NVMs. Although the term "mobile devices" may refer to many types of hardware with different architectures, usually those studies explore the characteristics of today's mobile phones and use common mobile phone applications' workloads to evaluate new solutions and techniques. Similarly to mobile systems, the

studies classified as embedded systems focus on NVMs energy efficiency, impact resistance, and small size. However, those studies could cover a larger array of applications and architectures, like wireless sensor networks and consumer electronics. Additionally, like in some general purpose studies, those studies usually assume limited capacities and high cost for NVM chips, since these technologies are still under development. Therefore, those works present some efforts to save physical memory, like avoiding duplicates through the memory hierarchy and providing compression. Finally, distributed systems focus mostly on the impact of NVM latency and persistence over the performance of compute-intensive distributed software and high-performance computing systems. Applications in this class include scientific applications and distributed file systems. The solutions explored by those papers usually explore the use of NVM to hide network latency, to improve the performance of distributed storage systems or to guarantee consistency among nodes. Solutions and problems addressed by those publications will be presented later in this document.

Other applications that, according to those papers, may also benefit from NVMs, but are less frequently targeted, include High Performance Computing (HPC), scientific computing, data intensive applications, long running systems and streaming applications. HPC and scientific systems are related to CPU-intensive workloads that, naturally, can greatly benefit from throughput offered by low latency persistent memories that may be placed close to the CPU. Data intensive applications include storage systems, databases and data mining applications, and are usually concerned with consistency, atomicity and performance optimization. These applications are very storage-dependent, and it is easy to see how they can be improved by NVM. Data-intensive and HPC systems are also usually associated with clusters and distributed architectures, inheriting the concerns of these classes as well. Long running systems are systems that are projected for high and long-term availability and should present a high grade of resilience and fault tolerance. In this case, NVM may be adopted to increase the amount of system memory and to reduce traditional complexity of moving data from operating memory to persistent devices. Streaming applications load large amounts of data in smaller chunks on demand (e.g. file transfer and streaming events, such as in IoT, security) and allow partial data to be processed before the entire object (e.g. file) is transfered. These applications usually work with sequential and predictable data accesses, which can be optimized for NVM with techniques like prefetching. Other applications include sensor networks [26], transactional systems [42], highly concurrent systems [76], semantic storage and application [1], high reliability (through data redundancy) systems [73], virtualized systems [74], and multidimensional storages [75].

This classification helps to understand the benefits and advantages of NVMs and to see these advantages from different perspectives. It also helps to understand the motivation behind the solutions proposed by those studies and, therefore, contributes to answering the research questions investigated in this work.

The next classification focuses on the problems addressed by each publication. The idea now is to identify the common problems impacting the use of NVM technologies as well as the methods used to address them. Naturally, the categories listed here are intimately related to the type of contribution categorized earlier in this work.

Table 3.4: Topics closely related to storage and file systems on NVM and the studies that explore them.

Problem	Studies
Access Interface	[89], [90], [114], [1], [126], [132], [92], [52], [51], [111], [43], [35], [82], [118], [57], [16], [95], [9], [97], [4], [56], [116]
Metadata Management	[70], [129], [1], [28], [119], [51], [29], [96], [111], [12], [55], [68], [43], [106], [13], [16], [80], [18], [75], [56], [107], [128]
Space Efficiency	[127], [26], [114], [21],[55], [68], [48], [80],[100], [9], [86],[94]
Garbage Collection	[27], [26], [126], [29], [96], [63],[54], [80], [36], [93], [128]
Atomicity	[37], [21], [132], [12], [83], [38], [16], [95], [110], [116], [128]
Write Amplification	[21], [82], [72], [45],[49], [16], [62], [78],[102]
Fragmentation	[70], [126], [76], [81]
Parallelism	[127], [52], [76], [110]
Transparency	[70], [55], [15], [131]
Mounting Time	[129], [51], [96], [128]

Therefore, Table 3.4 and Table 3.5 list the most common issues addressed by the analyzed studies. Table 3.4 shows topics that are considered to be concerns when NVM is used as storage. For this purpose, storage is considered whenever a structure, like file system or object store, is built over NVM and used to consistently store long term data and metadata. Thus, cases where NVM is used as operating memory, like current volatile memory, or used as buffer or cache, are not considered examples of storage. In Table 3.5 are listed topics that are not storage exclusive: they are also relevant when NVM is used for other ends like main memory or write buffer.

Both tables are ordered by the number of times each problem is addressed in the analyzed studies, in order to highlight the most frequently targeted problems in the area. This classification tries to maintain a balance between the generalization and specification of each problem looking to enumerate a reasonable number of categories, while also providing as much detail and information about the focus of each analyzed study. For instance, studies that focus on optimizing a file system's block allocation and studies that aim to avoid data duplication in the memory hierarchy could both be classified as software overhead studies, since these studies try to reduce software generated latencies that are traditionally used to improve disk-based storage performance. However, the approach followed by these two lines of research are very distinct and have different impacts in the overall system. Hence, in order to preserve these unique characteristics, they should be divided into two different categories. Each of these problems will be further detailed in future sections (Section 4.2 and Section 4.3).

The classification also shows a relation between the studies retrieved and the NVM-related problems they address. Many of these studies focus on exploring the advantages of NVM to improve performance, reliability and usability of existing architectures and applications. However, since the focus on this classification is to identify common problems that impact NVM usage, only the problems

Table 3.5: Topics that are not exclusive to file system or storage.

Problem	Studies
Consistency Guanratee	[26], [37], [21], [126], [32], [132], [119], [51], [29], [111], [69], [55], [15], [43], [82], [118], [67], [72], [57], [49], [83], [38], [13], [16], [44], [73], [95], [91], [113], [60], [62], [78], [105], [81], [112], [102], [94], [4], [108], [123], [107], [116], [128]
Endurance	[27], [127], [129], [40], [96], [12], [55], [82], [124], [2], [45], [49], [125], [75], [131], [112], [101], [102], [36], [93], [94], [123]
Asymmetric Latency	[27], [127], [85], [40], [96], [3], [2], [49], [88], [44], [64], [75], [120], [131], [112], [101], [86], [66], [123]
Persistent Cache	[127], [7], [132], [54], [68], [74], [3], [83], [64], [60], [30], [59], [17], [53], [41], [71], [46], [123]
Software Overhead	[90], [21], [126], [111], [43], [10], [42],[131], [9], [11],[102], [108], [56],[31],[107], [128]
Energy Efficiency	[127], [26], [39], [3], [88], [64], [77], [11], [59], [97],[123]
Block/Page Allocation	[89],[90], [40], [126], [92], [63], [106], [102], [36], [123], [128]
Cache Optimization	[27],[21], [63], [109], [100], [120], [19], [102], [71]
Cache Consistency	[61], [21], [32], [132], [63], [118], [42], [123]
Memory Protection	[89], [126], [32], [92], [15], [118], [38], [128]
Reliability	[69], [79], [38], [91], [113], [102], [4]
Data Placement	[27], [119], [88], [103], [131], [56], [46]
Data Duplication	[89],[61], [32], [92], [108], [107]
Scalability	[100], [56] , [128]

directly related to the characteristics of NVM technologies addressed by these works were considered. Architecture, application and scope specific problems were, therefore, ignored.

4. ANALYZING THE RESULTS

In this chapter the results of the presented systematic mapping are discussed and the research questions established in Section 3.3 are reviewed. The classification presented previously will be used to illustrate the answers of these questions.

4.1 What are the differences between disk-based and NVM file systems?

One of the first concerns that arises, when migrating from HDDs to NVM-based devices, is related to the processing overhead of the many different software layers that currently compose the storage subsystems [126] and [32]. This overhead is acceptable when working with disk-based storage, since seek operations in a disk are much slower than both main memory access latencies and CPU cycles, which means that the overhead of RAM and CPU (which is only a fraction of a disk seek) to minimize disk seeks is a necessary sacrifice. However, in the case of NVM, access latency is supposed to be very similar to regular DRAM latency. Therefore, NVM file systems are designed to reduce this overhead, usually using simplified data structures and policies and sometimes bypassing [10] [32] or completely abolishing some software layers, like block drivers, schedulers or page cache.

Another frequently discussed question is related to the access interface to these new memories. The POSIX file-oriented interface is certainly the main starting point. Although it is an important API for legacy systems, as well as for standardization of storage access, it seems unlikely that POSIX will remain untouched through the way of adapting today's computers for using NVM technologies. Since most NVM technologies are byte-addressable, many studies [97] [126] argue that the best way to access these devices is connecting them into the memory bus, giving the processor direct access to these memories through a memory-like interface. Some studies [55] [43] [118] propose a simplified and friendly interface to access NVM (usually based in a key-value object store) while providing the same functionalities and guarantees of regular file systems. Other studies [15] [90] [11], however, choose to access NVM using block granularity to maintain compatibility with legacy systems. In that approach, NVM may be either connected in the memory bus or into an I/O bus (like PCI) to avoid interfering with (or competing with) main memory related hardware, like the memory bus itself and the Translation Lookaside Buffer (TLB).

Connecting NVM to the memory bus and accessing it directly brings a few challenges like avoiding exposing data to stray writes and the lack of consistency with the processor cache. The former is related to the fact that, on today's hardware enforced page-based protection, bugs in the code of the user, the file system or even the kernel, may generate undesired writes in the NVM, leaving it in an inconsistent state. Given the persistent nature of NVM, a single stray write may corrupt the entire file system. This is not a problem in disks, since there are multiple layers in the memory hierarchy that are responsible to check for this kind of error before data is persisted in the secondary storage. The problem with cache consistency is due to the fact that to ensure atomicity

and consistency, the system needs a certain degree of ordering, for example to ensure that metadata is updated only after its corresponding data is already written. However, due to cache eviction policies, data may be transferred from cache to the memory out of order. Thus, a method to ensure ordering of writing from cache to NVM is necessary. Both these problems are further explored in the next section.

4.2 What are the challenges and problems addressed by NVM file systems?

A list of common problems addressed by studies in the area of NVM can be seen in Tables 3.4 and 3.5. In this section these problems and their impact over the system are detailed.

Consistency Guarantee and Atomicity

In order to avoid system corruption and data loss through multiple writes and erases, most file systems implement a mechanism to ensure full consistency. This means that data stored in a physical medium, by a file system, must be consistent at all time, in such a way that if a power failure or system crash occurs during a write operation, data will not be permanently lost. Common mechanisms of consistency guarantee include journaling, shadow paging, log-structured file systems and checkpointing. Although these techniques are very important in overall consistency maintenance, they are among the main sources of overhead in storage systems. For instance, the simplest method of performing journaling involves writing all data in a pre-allocated space in storage, called journal, before updating this data in the file system itself. This is known for being extremely inefficient, as every write issued to the file system incurs in at least two writes to the physical device (write amplification). Atomicity mechanisms are also closely related to consistency: they ensure that operations (like updates to user data, for instance) are either completely successful or completely ignored. However, atomic models, like transactions are not trivially applied to NVM as they were to disks and specially regarding scalability and overhead issues. Hence, many studies [43] [21] [55] [67] [116] [128] propose different methods to improve the performance of these mechanisms while also enforcing the consistency and security of the storage system.

Access Interface and Transparency

An issue that is extensively discussed in several papers is regarding the method of accessing and managing NVMs. Regarding NVM Interface, the most common question is whether NVM devices should be accessed as a block device (through a block driver, like current flash SSD and magnetic HDD), through main memory interface or through a new interface, like heap-based and key-value interfaces [43] [55]. Additionally, many studies propose approaches to provide users with the means of allocating and accessing persistent areas of memory, through, for example, system calls and programming libraries directives [35] [118]. Some of these studies seek efficient methods to support legacy applications (e.g. using POSIX file system operations) while also improving NVM

usage and leveraging its performance. They try to provide transparency for the upper layers of software, hiding NVM specific details and implementations.

Reliability, Memory Protection and Endurance

Another highly desirable storage characteristic is reliability. Upcoming NVM technologies present some limitations in the endurance and reliability aspects. As mentioned before, the limited endurance of NVM devices may cause some blocks to wear-out faster than others, since some data blocks are updated more frequently than others [106] [125] [55]. This results in the permanent loss of these blocks and the consequential reduced storage capacity. Reliability, as it is used in this survey, is related to loss of data caused by some physical failure in the device. For example, as memory chips' density increases, the possibility of operations performed in the device cells creating noises in other cells becomes more likely [79]. Other problems, related to bus communication, may become a threat to reliability as well.

Stray writes (e.g. writes performed on an invalid pointer, possibly referencing memory out of the process' address space) may be the result of bugs in the kernel or in system's drivers, and represent another threat to memory consistency [15]. Since data in NVM is persistent, it can be permanently corrupted by this kind of failure. Therefore, some studies propose different methods of memory protection to avoid improper access to memory pages[32] [15]. These methods can be implemented in software or use specific hardware.

Asymmetric Latency

Another common property shared by NVM technologies is the asymmetry in the latency of its operations [3] [49] [44] [75]. In these memories, write (and erase) operations are much slower than read operations. In some cases, like NAND flash memories, for example, where data cannot be updated in-place (blocks need to be erased before they are rewritten), the problem with slow writes is even more problematic. Thus, strategies must be adopted to avoid unnecessary writes to NVM or to reduce the latency of these operations. This property is a key factor in the design of most NVM file systems.

Metadata Management

Another critical point of optimization in file systems is related to metadata structure and management. It is known that access to metadata is intensive [16] and updates are very frequent. Furthermore, corruption in metadata may lead to the corruption of big portions or of the entire file system. Thus, it is highly desirable for metadata management to have low overhead while also providing an efficient structure to index data within the file system. Optimizing metadata for NVM file systems must consider many aspects, like the limited space of NVM devices, the inefficiency of write operations, the limited endurance of the pages and the byte granularity. Many studies [70] [129] [28] suggest the use of NVM as a metadata-only storage in hybrid memories, considering the

high cost-per-byte of pure NVM devices in the near future. This approach aims to leverage storage's overall performance without losing reliability and using small amounts of NVM.

Space and Energy Efficiency

Space and energy efficiency are among the first desirable characteristics that one might expect from a storage system, and are not limited to NVM-based storages. File systems and techniques that focus on reducing energy consumption and space allocation in storage are usually aimed for systems with limited physical storage and power supply, like mobile and embedded systems [26] [64] [77]. The most common methods of reducing space usage is by employing some compression methods or simplified metadata structures [55] [114]. Energy efficiency may be improved by eliminating unnecessary operations, especially writes and erases (e.g. using buffers or read-before-write methods), since these operations are more costly than reads [127] [59] [39].

Persistent Cache

Given the challenges of integrating volatile caches with persistent storage layers, some studies [7] [64] [132] [53] [74] suggest the use of NVM for caches and buffers. Since NVM does not lose data in cases of power outage, NVM buffers can be very useful in improving storage reliability and security, while also reducing the necessity of periodic flushes to long-term storages, improving system's overall performance. Studies that explore NVM cache management usually present metrics and policies to take full advantage of these persistent caches and buffers, while also studying their impact on the overall architecture.

Software Overhead, Block/Page Allocation and Data Duplication

Since, in magnetic and optical disks, sequential writes and reads are much more efficient than random operations, great effort is made by the file system to avoid fragmentation of a file and reduce disk seek times. However, while this additional processing generated by the file system is acceptable for disk-based storage (CPU and main memory operations are usually thousands of times faster than disk operations), since NVM latency is similar to current DRAM latencies, this processing overhead may significantly degrade overall system performance. Besides, for NVM storage, random access is fast and file fragmentation does not represent a performance penalty for itself, although highly fragmented data may become an issue due to complex management of fragmented files, buffering of multiple blocks and garbage collector's inefficiency when working with random data. Also, performing additional copies, for page and buffer cache for example, between NVM and main memory (DRAM) may also become a source of unnecessary overhead, since data can be accessed directly from the NVM device with DRAM-like speed. To address these questions, many studies try to identify and eliminate functions and layers of software that would no longer be necessary given the properties of upcoming NVM technologies [10] [9] [126]. Generally, these studies aim to reduce

software overhead, optimize block and page allocation and avoid unnecessary duplication of data through the memory hierarchy.

Garbage Collection and Write Amplification

As previously discussed, wear-leveling techniques are used to extend a device's lifetime. However, they may introduce the problem of additional writes, known as write amplification. Some studies explore methods to avoid or to reduce the occurrence of these additional writes, while still providing some wear-leveling mechanism. Another consequence of wear-leveling techniques is the usage of a garbage collector. Usually, to avoid data to be re-written in the same block and therefore reducing its life time, data is written out-of-place. This means that the update is actually written in a new block and the old version of the updated data is marked as invalid. The garbage collector is responsible for erasing data marked as invalid in order to allocate space for new data. This may, or may not, involve additional writes as well. Although this process is executed asynchronously (ideally when the storage device is idle), it has significant impact in the storage overall performance, as well as energy efficiency and is a point of optimization by itself.

Cache Consistency and Cache Optimization

Some studies [42] [32] [132] address the problems of the integration of NVM with other memory layers in the memory hierarchy, like the processor cache for instance. Like previously stated, when the NVM device is connected to the memory bus and directly accessible by the CPU, data may be written directly into NVM from the CPU's cache. Something similar occurs when NVM is employed as secondary storage and volatile write buffers are used to reduce the number of writes or hide latency. Since caches and buffers are volatile memories, data stored in cache lines may be lost upon system failures. Additionally, data must reach the memory in a consistent order so mechanisms like journaling can work correctly. Hence, some additional process is needed to ensure that cached data is correctly written back to the NVM.

While the aforementioned studies focus on providing cache and buffer consistency, others focus on tailoring buffer management and cache policies to NVM storages or main memory. These studies seek the optimization of cache and buffers using NVM-aware algorithms. For example, write buffers designed for disks and flash SSDs work with data in blocks and sectors since these are the units supported by the underlying persistent devices. However, NVM technologies (excluding NAND flash) can be accessed on a byte granularity and taking advantage of this granularity may significantly improve performance [32] [44]. However, for the system to benefit from this advantage, the buffer management needs to take this factor into consideration.

Data Placement

Another similar and common trend in hybrid memories is to use fast memory (NVM or DRAM) to store frequently accessed data. In this case, the file system needs a strategy of block

placement to decide whether data (pages, blocks or files) should be stored in the faster memory or in the slower, denser and, presumably, persistent memory [131] [103] [27]. This strategy must identify critical data and determine whether they are temporary or must be persisted.

Fragmentation

As mentioned in 4.2, although fragmentation in NVM does not directly result in performance penalty, it may be a problem in some cases. For instance, scattering multiple pages of a file through different erase blocks in NAND flash may cause an increase in the number of block merges [70]. In other cases, dealing with larger page sizes (larger than the traditional 4 KB), or even segments, may cause problems of internal fragmentation [126].

Mounting time and parallelism

One of the problems addressed by traditional flash file systems is regarding the file system mounting performance. On one hand, since NVM technologies have endurance limitations, keeping frequently updated data, like superblocks and inodes, in a fixed position in the device may not be a good idea. On the other hand, since NVMs may, eventually, scale up to petabytes, scanning the whole NVM device is not an option either. Thus, reducing scan time, the scanned area and memory footprint may be somewhat challenging especially in log-based file systems [51]. Furthermore, the growing parallelism in SSDs architectures is rapidly increasing, greatly improving the throughput of NVM-based block devices. However, traditional file systems usually fail to take full advantage of such parallelism wasting valuable resources [127].

Scalability

One of the main advantages of novel NVM technologies over current volatile memories is their superior density. Allied with its low energy consumption, this factor allows systems to have large amounts of main memory (e.g. petabytes). However, managing such a large main memory is not common in today's architectures and, in order to do it in an efficient way, some challenges may need to be solved. For instance, addressing in such memory is not trivial. On one hand, issuing addresses on a page granularity (which usually range from 4 to 16 KB in current systems) may be inefficient as too many addresses may take a toll on address translation performance [100]. On the other hand, using larger or multiple page sizes or even segmentation may cause problems like internal fragmentation, write amplification, protection issues and drastically increase memory management complexity.

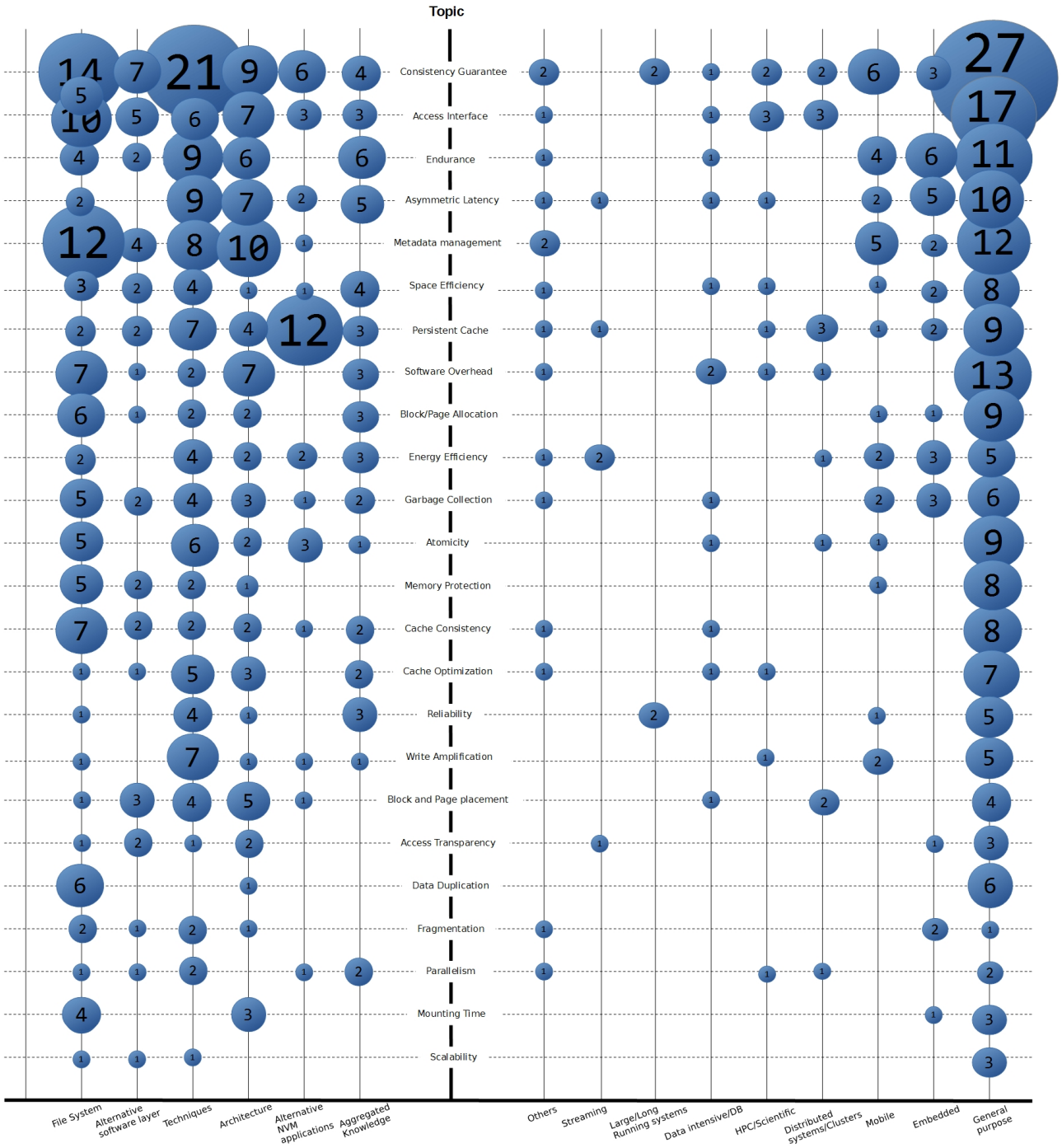

Figure 4.1: Bubble plot illustrating the focus and distribution of research on the field

4.3 What techniques and methods have been proposed to improve NVM file systems?

In last few years, many different mechanisms and solutions have been proposed both to explore utility and performance of NVM technologies and to mitigate its weaknesses. Figure 4.1 helps us to visualize how the research effort in the area is currently distributed and which problems

discussed previously have received most attention from researchers. The tendencies shown in Figure 4.1 are also useful to identify poorly explored territory within the NVM area and possible opportunities or niche areas for study.

In this section we do not present the techniques and methods directly, but rather the problems as discussed previously and then regarding each problem, we discuss the technique or method that was used to solve it.

Consistency Guarantee and Atomicity

Relative to the file system consistency, traditional mechanisms, like journaling and shadow paging are tailored to obtain improved performance in NVM. An example of fine-grain (e.g. 128 bytes per log entry) metadata journaling is presented in FSMAC [16]. FSMAC creates multiple versions of the metadata before committing data to in-place update, enabling the operation to be undone. PMFS [32] uses a similar method of fine-grain journaling for metadata while also using copy-on-write for file data pages. In [44] a two-level logging scheme is presented, mixing a fine-grain journaling mechanism with a log-based file system structure. In-memory Write-ahead Logging (IMWAL) [105] involves reusing the data log to update the database file by remapping memory addresses, thus reducing the number of writes required for a commit operation. It also proposes to make the in-memory file system the only responsible for journaling, avoiding the "Journaling of Journals" issue when both database manager and file system perform logging separately.

The Byte-addressable Persistent memory File System (BPFS) [21] uses a tree structure of indexes for its files. Metadata is updated in-place through atomic updates (up to 8 bytes), therefore no additional writes or data copies are necessary in this case. For larger updates, BPFS uses an extension of the shadow paging technique: it performs a copy of the blocks being updated, update the necessary data and write these copies in new blocks. The main drawbacks of this solution is that 8-byte atomic writes support is mandatory and updates may cause a cascade update in large or sparse writes, generating extra copies and write amplification.

On the other hand, on Kiln's [132] approach for consistency data is written to a persistent cache before it reaches the long term NVM storage (accessible through the memory bus). In this design, data from volatile cache lines is written to a NVM cache in a transaction fashion. When the transaction is committed, the remaining volatile cache lines belonging to that transaction are flushed to the NVM cache and the transaction is committed. When cache lines are evicted from the NVM cache they are copied back to the NVM storage. This approach also needs additional hardware and modifications to the cache controller.

A file system mechanism that is particularly susceptible to consistency problems is the file mapping. File mappings is a popular method data access model which is also the base for other storage systems like Mnemosyne [118]. While traditional mapping mechanisms require data to be copied back and forth from the disk to DRAM, NVM file systems like PMFS [32] allow applications to access NVM pages of mapped files directly by simply mapping them in the process address space. However in both cases, the application has no control over the order in which data

is made persistent, which may cause inconsistencies within a file's data. With that in mind, a few studies propose making these mappings perform updates in the file atomically [95] [116] [128]. Even though the implementation of the solutions may differ, all studies seem to agree on giving the user the control over when the data written in this mappings are made durable (through *fsync* and *msync* calls, for instance)

Access Interface and Transparency

In order to provide transparency and a rich interface to access NVM devices, Muninn presents an object-based storage model. In this approach, data and metadata management is delegated to an object-based storage device (analogous to a block device), that exports an object interface. This interface provides basic operations to manipulate variable-size objects, like reading, writing and deleting objects from the system. For more flexibility, Muninn allows file systems to access the object-based storage device functionalities in a publish-subscribe fashion, enabling the object-based devices interface to be extended, offering device-specific operations to the system. In this model, to maintain compatibility with legacy systems, an object-based file system can be used to provide a POSIX interface to applications and operating system. This file system is then responsible for translating these POSIX-format requests to object semantic and to call the appropriate object-based device operation.

SCMFS [126] reuses Linux memory management module to work with NVM. This allows SCMFS to work with a contiguous address space, using memory page mapping mechanisms to translate logical addresses to NVM physical addresses. SCMFS also extends the memory management interface to allow users to allocate and map NVM space. For example, a new function nvmalloc is designed to allocate a contiguous space in NVM virtual address and two functions *nvmalloc_expand* and *nvmalloc_shrink* are included to respectively increase the size and release unused space of an allocated NVM space. There is also a *clflush_cache_range* function that combines the features of the *cflush* and *mfence* operations to ensure the persistence of critical data by avoiding retention in volatile caches.

Mnemosyne [118] grants users access to NVM by persistent regions. Persistent regions are segments of memory that can be manipulated like regular memory by user code but are allocated in NVM. In order to allocate these persistent regions, users can either mark a variable as *pstatic* or calling the *pmap* function. These persistent regions are stored in files (that may be kept in a secondary storage, like a SSD) and mapped (through *mmap* system call) by demand to the NVM when their owner process is started. Additionally, Mnemosyne offers the user the possibility of persisting data atomically through durable memory transactions. Users can mark blocks of code with the atomic keyword and Mnemosyne will ensure *ACID* (Atomicity, Consistency, Integrity and Durability) properties of all changes made to persistent data inside this block of code through a transaction structure.

Reliability, Memory Protection and Endurance

To improve reliability of operations performed in NVM, error-correction codes are usually the solution [102]. These codes are used to verify the integrity of data after performing a physical operation, similar to the use of Cyclic Redundancy Check (CRC) and checksums in network communications. A higher level solution for reliability is the file system snapshot mechanism. Snapshot techniques store older versions of the file system data and metadata ensuring its overall integrity through redundancy [69]. As for memory protection, a simple and common method is to mark pages as read-only while mapping them into virtual address space and marking only the pages to be updated as writable when a write is issued. One possible solution [38] is to explore a combination of the EVENODD error correction code algorithm and memory protection to improve the security of storage systems. In that solution, pages are locked for read-only purposes, and are only unlocked when a write is explicitly issued. PMFS [32] does something similar to ensure memory protection, avoiding corruption by stray writes. Mnemosyne [118] avoids that memory leaks generated by a program corrupt the system or other programs by separating the virtual persistent memory regions of each program and storing them into different files.

One of the most problematic and intensively studied limitations of NAND flash memory is regarding its limited endurance [93] [2] [125] [82]. Much of this knowledge can be applied to byte-addressable NVM as well since, in their current state of development, they suffer from the same limitation. However, techniques somewhat differ from flash to Phase Change Memory (PCM), because, for example, PCM can be written in-place and in terms of bytes instead of pages or blocks. In order to improve the lifetime of both these technologies, two approaches are usually explored: reducing the total number of writes that reach the physical device [82] [127] and using wear-leveling techniques to distribute writes and erases equally over all the devices blocks in order to delay the wear-out of physical structures. Wear-leveling techniques can be separated in two types as well [124]: techniques that try to spread writes and erase equally over the entire memory's surface and techniques that try to identify hot and cold files and blocks in order to store more frequently accessed data in less worn-out blocks.

The W-Buddy [106] is a wear-aware memory allocator that extends the Buddy [50] memory allocation technique to consider the endurance of memory pages and to provide wear-leveling. In this technique, memory is organized in a binary tree-like structure, where each level contains a different size of memory chunk, where the root node contains chunks of N bytes (where N is the size of the biggest chunk allowed) and the second level contains twice the number of chunks presented in the root, and each chunk is (N/2) bytes long. Each chunk stores a counter S representing the number of times that the chunk was updated and a bitmap used to identify free sub-chunks in the lower levels of the tree. When the system needs to allocate a chunk of a certain size, the W-Buddy allocator starts the search for the best fit by looking at the root of the tree and run through the levels using the S counter and the allocation bitmap to locate the less worn-out chunk available.

The Differentiated Space Allocation (DSA) algorithm [45] is a wear-leveling technique designed for PCM that focus on providing a method of increasing a device's lifetime with low

overhead and without the necessity of a garbage collector. To this end, DSA uses over-provisioned space (hidden from the operating system) to write frequently updated data, and only maintains an update counter for recently accessed chunks of the memory. When the update counter of a chunk reaches a determined threshold, a new chunk from the over-provisioned space is allocated and the old chunk is marked as expired. The same happens with the new chunk if it also reaches the threshold. When all chunks from the over-provisioned space are either allocated or expired, DSA allocates new chunks to be used as additional area and free the older expired chunks. The chunks to be used by DSA as over-provisioned space are selected randomly in order to avoid the time and space overhead of looking for the less updated chunks.

In [124] and [125] two different heuristics to identify frequently updated (hot) files are presented. For instance, files that are marked as read-only, files belonging to specific users that rarely log into the system, read-only operating system files and files that are rarely accessed or are related to processes that are rarely run, are good candidates for cold or frequently read files. On the other hand, database files and logs are good example of hot data. In [125] these heuristics are applied to the Android operating system, exploring the way this operating system organizes files in the storage and specific metadata, like the application a file belongs to.

Asymmetric Latency

The next subject is the asymmetry between write and read latencies that NVM technologies present. A common way to deal with the high cost of writes is by simply employing a DRAM either as a write buffer or as a part of the main memory composing a hybrid memory layer. In a hybrid memory model (DRAM and PCM) [101], techniques like lazy and fine-grain writes could reduce the amount of writes to reach the PCM device and to improve its lifetime. The results show significant performance improvement (up to 2 times faster in some cases) over a purely PCM memory, while adopting small amounts of DRAM.

Another approach [3] explores the fact that the amount of energy and time needed to make a write to the physical memory is directly proportional to the duration of the persistence. This means that higher latency and energy dissipation is needed to make data durable in NVM for longer periods. However, some portions of data are updated very frequently (e.g. file system metadata) while others may belong to temporary files. In this case, the persistence process can be relaxed, reducing the durability of data but also the latency of write operations. The system can distinguish hot and cold files and apply different write intensity in each one, which, since hot files usually dominate the accesses to storage, may improve performance as well as energy efficiency significantly.

Another interesting method to reduce write latency is the read-before-write technique [77]. In this technique, before a page of data is updated, the bitwise difference between the old and the new data is calculated. The device then proceeds to change only the necessary bits in the updated page, reducing the amount of work performed and, consequently, the latency of the write operation. The study further improves this technique by locating free pages that contain bit values similar to

the data being written. It keeps small bit samples for each free page, uses a specific hardware to make bitwise comparisons to the pages being written and select the page according to the grade of similarity.

Metadata Management

Due to its byte-accessibility and low latencies, multiple studies propose the adoption of NVM in metadata management and storage. FSMAC [16], for example, proposes an architecture where DRAM-based main memory is expanded by NVM. At runtime, metadata is loaded from the disk into NVM and is accessed through a byte-addressable interface, just like it happens in current systems where metadata is loaded into DRAM before being accessed. FSMAC manages NVM separately from DRAM and allows metadata from multiple file systems to be stored into the persistent memory region at the same time. The fine grain log mechanism, mentioned earlier, is used to maintain the integrity of this metadata across multiple updates. Since metadata is loaded into NVM once and only written back to disk at the time of unmount, this approach greatly reduces the number of effective write operations to reach the slow long-term storage.

Muninn [55] explores the byte-addressability and random access speed of NVM to implement an object and hash based system structure. It uses hash functions and bloom filters to address key-value pairs, eliminating additional metadata and address translation information. LiFS [1] also exploits the advantages of persistent memories, in this case to expand the capabilities of metadata in order to store more meaningful information in these structures. It proposes a model of extensible metadata that allows the creation of links and relationships between multiple files and custom attributes. This additional information can be used by operating systems and applications for the purposes of indexing, semantics analysis and optimization.

In another approach [107], the file index metadata is designed to be similar to the kernel's page tables. This approach allows the user process to map a file into its address space by simply adding a single entry in its page table. This approach also eliminates the overhead of faulting pages into the user's process address space. When the process halts or the file is unmapped, the file address space is safely released by the operating system. In this case, metadata is not designed to leverage NVM byte-addressability or to be space efficient, but rather to make access to files easier.

Space and Energy Efficiency

In general, to improve space utilization in a NVM storage, studies propose the use of compression techniques and simpler data structures. Muninn [55], for example, uses bloom filters, that are data structures known for being space efficient, to reduce the amount of stored metadata. MRAMFS [114] is a file system designed to be space efficient. It compresses file data as well as metadata, trying to balance the tradeoff between access performance and compression level. A different approach is given by the dynamic over-provisioning technique [48], that focus on efficiently using the additional space inside flash SSDs, which is used to allow efficient out-of-place updates and

garbage collection (over-provisioning). This additional space is used by the dynamic over-provisioning technique to store temporary data in the cases of high storage demand.

In the case of energy efficiency, optimization is usually made in terms of reducing the amount of data written into the physical device [127]. This is because the amount of energy necessary to perform a write operation is much higher than the energy necessary for a read operation. A simple method uses a small portion of volatile memory as write buffer, reducing the number of writes that reach the persistent device. However, volatile memories like DRAM and SRAM usually consume more energy than NVMs while they are not being accessed. In another method [64], NVM is used as an instruction cache in order to leverage performance while also improving the persistent memory's lifetime by storing only read-intensive data (in this case, instructions). Copy-before-write can also be used in order to change the minimum number of bits in a page [77]. The algorithm search, among the pages stored in the device, for pages similar to the one being written. When a page is selected, only the divergent bits are flipped and, therefore, less work is performed.

Persistent Cache

Many papers study the impact of NVM-based buffers and caches and propose optimizations for these memories. For example, four different eviction policies are proposed to be used with a persistent NVM-based write buffer for SSD storage [54]. These policies are optimized for flash memory storage and are FTL (Flash Translation Layer) aware. Therefore, the policies aim to explore temporal locality while favoring sequential writes to the flash device (e.g. by clustering sequential pages and flushing them altogether to the SSD) and avoiding the costly merges made by FTL's garbage collector. The study [54] also presents the design of a simplified FTL design optimized for these buffer policies.

TxCache [83] employs NVM as a disk cache and provides a transaction model to allow consistent and versioned writes to this cache. In order to support transactional semantics, TxCache exports an extended SSD interface offering methods to, for example, start, commit and rollback a transaction (BEGIN, COMMIT and ABORT). Writing data into the TxCache device using transactions guarantee atomicity and consistency of write operations, because TxCache always keeps a backup of older version of the pages being updated by the transactions in the disk. Furthermore, pages being updated through transactions (in the disk cache) will only be written back to disk once their corresponding transactions have already been committed. This guarantees that, if a transaction fails for some reason (e.g. system failure or power outage), either the most recent version (in disk cache) or the older version (in disk) will be available for recovery. TxCache also keeps track of all pages updated by a transaction using page metadata, which speeds up the system recovery process in case of crashes and may also help to enforce sequential writes to the disk.

The persistent processor cache scheme [132] uses something similar to TxCache transactions. Data is transferred from volatile to non-volatile cache in a transaction fashion and, once the transaction is committed, the cache controller flushes all dirty data relative to the committed transaction that still resides in the volatile cache to the persistent cache. Once all data is written

to the non-volatile cache, data is considered persistent. The same assumption made by TxCache is used: NVM-based cache will always have the most up-to-date data, while the storage have an older copy, and, therefore, no additional consistency mechanism is needed. Once the transaction is completed, cache lines from the persistent cache are allowed to be written back to the NVM storage by the cache eviction policies.

NVM can be used also in combination with regular volatile memory to compose a hybrid page cache [68]. In this type of architecture, volatile memory is used to speed up the access to frequently accessed blocks and reduce the number of writes that reach the storage. The NVM, on the other hand, is accessed in a byte level and is used as a cache buffer, storing only updated fragments of data as a backup for the storage in the case of critical failures. Data is written back to the storage device when selected to be evicted from either volatile or non-volatile cache. Something similar is proposed elsewhere [44]. But, as mentioned earlier in that survey, in this case, instead of updating data in-place, the two-level logging scheme merges modified data with unmodified data in the data block in the long-term storage. This process creates a new version of the modified data block and invalidates the older version, avoiding partial writes and improving the system's reliability. In both these approaches, cache write-backs and flushes can be greatly reduced while reliability and consistency is guaranteed.

Software Overhead, Block/Page Allocation and Data Duplication

Much of the performance degradation of NVM devices in traditional systems is the result of file system and operating system software overhead. Thus, minimizing software complexity is another point of optimization targeted by studies in the area. Moneta-D [10] architecture moves file system's and operating system's permission checks and policy enforcement to hardware. This architecture also allows accesses to NVM to bypass I/O schedulers and avoid context switches. Data is accessed through channels provided by a user space implementation driver library that manages the low-level mechanisms of Moneta-D and offers a POSIX-compatible interface for legacy systems. The Moneta-D architecture also offers high parallelism through the replication of hardware and memory controllers.

Another file system proposal [108] relies on user-space code to avoid the overhead caused by the I/O stack of the operating system. Building a file system in user level has some key advantages to that end, like minimizing the number of context switches, bypassing unnecessary kernel space layers and allowing more fine-grained data management. The file system further improves the performance to access data by keeping the file mapping overhead to minimum by employing the file virtual address space framework [107]. In this framework, every file has its own address space, as they are composed of an index structure that mimics the structure of the kernel's page tables. Therefore, when an application maps or open a file, all the system has to do is to add a single entry into the highest level of the application's page table, pointing to the file's address space structure. Hence, the file is mapped into the process address space, and the file's pages may be accessed directly with no further paging.

Another way to improve file system performance is by optimizing block allocation and deallocation policies. SCMFS [126], for example, uses space pre-allocation, preemptively allocating null files and allowing existing files to keep extra space allocated. This mechanism makes the file creation process, as well as future data updates and appends, easier. The W-Buddy [106] algorithm, presented previously, organizes free memory space in a structure similar to a binary tree, and uses allocation bitmaps and update counters in order to find the best fit when allocating new portions of memory. The binary search is significantly more efficient than a sequential search, reducing the time necessary to find a page to allocate, but also allowing the location of ideal fits.

The problem of duplication of data across NVM and DRAM is yet another process that remains from the traditional storage model. To address this problem, PMFS [32] integrates the execute-in-place (XIP) [117] functionality that allows data in persistent storage to be accessed directly. XIP bypasses the page cache and I/O scheduler, eliminating unnecessary duplicates of data from the NVM to DRAM. The same applies to mapped files: page descriptors are allocated and point to the mapped file in the NVM storage, but no pages are actually copied to the cache. Data is directly placed in the process user space in order to be accessed and updated by the processor.

Garbage Collection and Write Amplification

Regarding the common methods used to address the ensured consistency and endurance problems, minimizing the write amplification is usually one of the main goals aimed by the studied techniques. OFTL [82] uses page level metadata and reverse indexes in order to allow the tree structure of objects to be recovered in case of corruption of the indexes. In OFTL, each object contains a tree structured index where the leaf nodes contain the pages with the object's data and metadata. Each page contains data about which object they belong to and the offset inside the object. Therefore, even if the object's index structure is corrupted, it can be recovered using this information, hence, eliminating the need of journaling or shadow paging. To reduce the effort that would be needed in order to scan the device for these pages and recreate the indexes, a window containing the most recently updated blocks is maintained by OFTL. OFTL further reduces the number of page write operations by grouping multiple small writes (smaller than a page, for that matter) in a single page before effectively writing them into the device.

The fine-grain log mechanism used to maintain consistency by FSMAC [16] is an approach to minimize the impact of write amplification in the file system. By writing only the necessary metadata to the log, FSMAC reduces the overhead versioning and the amount of data replicated by additional writes. The differentiated space allocation presented earlier, uses the reverse index mechanism and over-provisioned space in order to eliminate the necessity of journaling and garbage collector. Since the mechanism allows a chunk of physical memory to be overwritten in-place a certain number of times (threshold), it reduces the replication of data using copy-on-write techniques like other wear-leveling mechanisms. Another technique called Delta Journaling [62] reduces the number of writes required by persistent memory file system logging by means of storing the delta of the changed blocks when a high compression ratio can be attained. Similarly, FCKPT [78] proposes

using only the delta of modified memory pages for the persistent memory checkpoint file of High Performance Computing applications.

The garbage collector is a mechanism introduced by flash-based SSDs that is responsible for erasing invalid blocks, merging blocks and allocating free space. ELF [26] presents a simple implementation of a garbage collector. When the number of free pages reaches a determined threshold, the cleaner process is started. The cleaner is responsible for identifying and erasing blocks belonging to deleted files, as well as updating indexes and bitmaps. It also may perform merge between partially valid blocks (blocks that contain some pages marked as invalid) to free additional space. DFTL [63] optimizes the garbage collection process by avoiding fragmentation and enforcing sequential writes, reducing the occurrence of costly merges between partially valid blocks. It also allows the free pages threshold (that determines the frequency of garbage collection execution) to be tuned according to the system's workload. The NOVA log-structured file system [128] employs two garbage collection algorithms: a fast light-weight one used to free pages composed exclusively by old log entries and a slower thorough one that may perform merging operations, copying valid log entries from multiple pages in a new page.

Cache Consistency and Cache Optimization

The problem of consistency in file system may face some additional challenges when it comes to working with the processor cache. Since these caches are volatile and hold data for a significant amount of time, in cases of power failures, data that are assumed to be in the persistent memory may be lost. This may become a serious reliability issue. The most simple way to deal with this is by simply avoiding cache (write through). However, this may have serious performance impact in the overall system. Additionally, there is also the problem with data reordering, mentioned earlier.

The most common method for dealing with these problems is by issuing (explicitly or not) barrier and flush commands to the cache in order to ensure the persistence of cache lines and the order in which data will be written to NVM. SCMFS [126], HEAPO [43] and PMFS [32] use a combination of *mfence* and *clflush* to ensure ordering and consistency. The *clflush* operation forces the eviction of cache lines, invalidating them and causing them to be written back to the memory. The order in which data will become visible in the memory may be defined by barriers through a memory fence instruction (*mfence*). The *mfence* instruction is a barrier that ensures that all operations prior to the *mfence* call will be performed before the instructions that follow the *mfence* call. The increased traffic caused by periodic flushes plus the overhead of the ordering instructions may significantly degrade performance [132].

To avoid both periodic flush and write-through methods, BPFS implements the concept of epoch barriers. In this approach, additional control is added to caches in order to organize cache lines in epochs. The epoch barriers are issued through an epoch instruction, just like with *mfence*. In the cache, each cache line is marked as belonging to a specific epoch and each epoch have a precedence (through a sequential number, for example). In this case, if the epoch A precedes the

epoch B, the cache lines belonging to the epoch B can only be evicted if every cache line from A has already been evicted. Thus, cache is not flushed and data is only written to persistent storage when evicted, reducing memory bus traffic. Another way to reduce flush to memory is the persistent processor cache [132]. Since the NVM cache is physically close to the volatile cache and does not compete for the memory bus, there is no need to explicitly issue an ordering instruction when writing between volatile and non-volatile caches [132].

Data Placement

Another architecture design trend involves the combination of NVM and DRAM or SRAM in a single, hybrid memory. Because of the many differences between these two types of memory, a block placement policy is needed for the system to be able to determine in which memory it would be more advantageous to place each piece of data. One strategy could be an architecture that features an NVM/DRAM hybrid cache [103]. In order to optimize the utilization of this hybrid cache structure, the Rank-aware Cooperative Caching (RaCC) block placement algorithm is proposed. This algorithm uses multiple queues with different priorities to store descriptors for each cached object. These descriptors are used to keep track of the number of times an object in the cache was referenced and when it was last accessed. RaCC prioritizes the allocation of volatile memory (DRAM) for the most popular objects, since DRAM is significantly faster than current NVM technologies. Thus, when the number of accesses to an object cached in NVM reaches a certain threshold, it is moved to DRAM and vice-versa. Objects can be released either when the cache is full and new objects need to be cached or when the system detects that an object was not accessed for a certain amount of time.

In a more generic implementation, the algorithm shown in [27] uses dynamic programming to track the cost of each cached data block for each type of cache. This includes the cost of moving blocks through different caches. The algorithm then uses a couple of heuristics to minimize the overall access cost and to find the optimal placement for every cached block. The algorithm in terms of time and space is polynomial. Conquest [119] chooses a simpler approach: since, statistically, most accesses go to small popular files and most of a system's storage space is consumed by large files, Conquest simply stores large files (larger than a predefined size) into disk and small files and metadata into NVM. This results in a significantly more simplistic mechanism than block placement algorithms, which also means less overhead in NVM management.

Although no specific algorithm for data placement is proposed by pVM [56], the overall structure adopted by the persistent virtual memory model makes page allocation and data placement across different memory technologies significantly easier. pVM treats NVM as a NUMA node in order to both account for the difference in bandwidth between DRAM and NVM and also to make the transition to the pVM model simpler. This design allows pVM to be extended in order to support more sophisticated data placement policies sensible to different NVM technologies.

Fragmentation

The hybrid FTL design presented in [70] employs byte-addressable NVM to store important storage metadata, like allocation bitmaps and mapping tables. These structures provide information about the status of physical blocks and space usage. The FTL then uses the information to avoid placing logically related pages in different erase blocks, prioritizing keeping related pages contiguously. Additionally, since structures that are traditionally stored in volatile memory, like page mappings, are kept in persistent memory, they do not need to be frequently flushed to be consistent, increasing both performance and robustness of the storage system. The adoption of superpages in SCMFS [126] (see 4.3) is an example of mechanism that can lead to internal fragmentation. In this specific case, in order to reduce the impact of internal fragmentation, regular sized (4 KB) pages are used for more fine-grain data. This solution adds complexity to the page management (depending on implementation) although it still may gain in performance due to the improvement on the address translation process.

Mounting time and parallelism

An efficient (and relatively simple) way to deal with the mounting time constraint is proposed by FRASH [51] and PFFS [129]. Both are systems designed for hybrid storage models that use byte-addressable NVM to store metadata and indexes while using NAND flash for general storage. This approach has many advantages: the scanned area is much smaller (since the amount of byte-addressable NVM adopted is much smaller than the amount of flash), the scan speed is much faster due to byte-addressable NVM low latency (compared to NAND flash) and memory copies may be reduced, since metadata can be directly accessed in NVM. Regarding parallelism, PASS is a scheduler developed specifically to take advantage of SSD architectures. To do so, PASS divides the storage in scheduling units in order to avoid interference caused by concurrent operations.

Scalability

In [100] a problem with SCMFS design is described, where access to the file system's pages pollutes the TLB which increases the number of misses during address translation. To alleviate this problem, SCMFS employs an alternative type of page, called superpage, that is 2 MB long, thus reducing the number of addresses needed to designate large portions of data. These superpages are used to address large files, while regular pages are used for the remaining data, in order to maintain space efficiency. SCMFS initially allocates normal pages for every file, and upgrades to superpages as the size of the file increases. It is a relatively simple solution that may compromise space efficiency in a small scale, may cause write amplification and adds complexity to the file system for the sake translation performance.

The NOVA file system [128] addresses scalability by employing per-CPU metadata. Each CPU has its own journal and inode table in order to leverage the system's concurrency and avoid locking. The CPUs also have separate free-page lists used by the memory allocator mechanism.

Whenever a new page is needed, the CPU first tries to allocate it from its own page list before resorting to other CPU's free pages. It is important to note that, naturally, locking mechanisms and critical regions are still used in NOVA, however the file system is built to keep process locking to a minimum, exploring the advantages multiprocessors.

The study in which pVM is presented [56] criticizes the usage of VFS as basis for future NVM storage due to its scalability and flexibility limitations. According to the study, the main issues are that persistent memory cannot be transparently allocated when NVM is exposed as a file system (e.g. during memory pressure times) and that file system operations (e.g. updating metadata and logging) significantly increase TLB and cache miss ratio. Given these limitations, pVM chooses to extend the kernel's virtual memory subsystem, allowing applications to allocate persistent memory through an API extension of the existing VM API (e.g. malloc and free functions). In this design, NVM may be allocated just like regular DRAM and application virtual memory may be transparently allocated from NVM when the system is in DRAM shortage. For data storage, with a semantic closer to that of a file system, pVM also provides an object store that is accessible through a user-level library.

4.4 What is the impact of new file system models on the overall architecture?

Many of the studies analyzed and discussed by this work explore alternative approaches to optimize the usage of NVM devices. Although the architecture of current systems may greatly benefit from upcoming memory technologies by simply replacing disks for NVM-based devices in their storages, it is clear that such approach presents many limitations that prevent systems from exploring the full potential of the new memory technologies. Adjusting the current architecture to accommodate NVM devices must take in consideration many different aspects of both current hardware and software. Additionally, it must be considered that, given its limitations (especially in density, compared to disks), current NVM is not yet ready to replace HDD, and this will be the reality for a long time. Thus NVM devices are supposed to close the gap between HDDs, SSDs and main memory providing both low latency and persistence.

Regarding the NVM access, the simplest method is to access it through a block driver and mount a regular file system over it. Modifications in software and hardware necessary are much less dramatic in this case and legacy applications can enjoy the high speeds of NVM without any further optimization. Alternatively, a NVM-specific file system may be mounted over a NVM device, in order to explore its advantages and overcome its deficiencies while also providing a POSIX compatible interface. In this case, while the file system can maintain compatibility by presenting a standard interface, it may also expand this interface providing optimized functions. Finally, NVM can be accessed through a memory-like interface (e.g. pmalloc, pmap) or through more user-friendly interfaces provided by NVM-specialized libraries and persistent heaps.

Giving the CPU direct access to NVM by connecting NVM device into memory bus can cause a significant increase of data traffic in the memory bus and concurrency for memory resources like mapping tables [126]. Also, reliability and endurance problems may arise, especially if NVM is used as main memory, in which case access should be much more frequent compared to storage access. Hardware mechanisms to provide error control, memory protection and permission checks can eliminate much of software complexity and overhead, thus improving overall performance.

The study presented in [4] shows a prediction of the impact, over the operating system, of adopting NVM as storage, main memory or integrating both in a single memory layer. It discusses some interesting topics, like the implication of persistent application faults and errors and data portability. The latter refers to the fact that if architecture specific and kernel internal memory structures are used to maintain persistent data, then porting data from a system to another becomes more complex. This may also represent a challenge when working with multiple versions of a same system. The non-volatility of data may also introduce new exploits and security breaches, like cold-boot attacks. Some other characteristics of NVM technology may also be exploited by malicious software, like, for example, the limited endurance of devices. In general, the study shows that the adoption of a single layer of persistent memory serving as storage and main memory could vastly simplify memory management by eliminating concerns like page swapping, multiple address spaces and page caches. But it would also introduce new problems, like reliability, portability and security.

However, it seems unlikely, at least for the near future, that NVM will replace DRAM as main memory, but, instead, used in cooperation with it in a hybrid memory layer. In fact, many studies already study the possibilities of exploring hybrid memory. Contributions in this area includes data placement algorithms [27] and methods to integrate the file system and memory management [89] [90]. These are important steps to optimize the usage and explore the advantages of both volatile and persistent memories

Finally, foreseeing the limitations of NVM for the near future, multiple studies [129] [38] [16] [30] propose the use of small amounts of NVM, using them as write buffers or metadata-only storage, supporting a larger storage based in NAND flash or magnetic disk. This is an interesting way to improve storage performance and endurance while employing a reasonable amount of NVM. Other studies [70] explore the usage of small amounts of NVM to store only frequently accessed blocks of the disk/SSD, just like a cache. In yet another example, [64] presents a model of persistent instruction cache. In this case, the focus is on reducing the energy consumption for very low-power systems, which is achieved by storing read-intensive information (instruction loops) in NVM, avoiding the high cost of writes and the idle dissipated energy of DRAM.

5. ANALYZING THE STATE OF THE ART AND DISCUSSION

This section seeks to present an overview on current solutions and implementations of file system designed for NVM and on the main trends that seem to be the main target for future studies in the area. This overview is based on the patterns identified in the analyzed studies and on recent NVM-related technology advances and projects. The goal in this step of the study is to identify areas of the NVM-based storage that may need more attention (that may represent research opportunities) as well as the areas that are already saturated with research and that (according to the retrieved studies) already seem to have a few concrete solution models.

Table 5.1 shows different storage solutions that target NVM and the issues they address. It provides a simple overview and also a comparison between the main studied storage systems, enumerating the problems that each of them addresses. The last column of the table presents a simple conclusion about the research development of each specific topic. It helps to illustrate the current level of development of the NVM-based studies. The table also shows which topics need further exploration and which topics have been extensively studied and that have solutions proposed in the literature that seem to be accepted as good solutions.

Advanced: these issues have been targeted by a significant amount of research and some solutions have already been proposed and implemented. Most solutions to these problems seem to follow the same patterns and employ the same ideas, creating a clear model around which a concrete and robust mechanism could be built. It means that at least one existing solution could be adopted in an existing storage system and present satisfactory results.

Non-critical: the issue is either expected to be solved when the technology reaches a certain level of maturity, or it simply does not represent a threat in the architecture specified in Section 5.1 These issues may eventually become points of interest, as NVM-based systems evolve and change, but for the foreseeable future, it does not seem they are critical issues.

Lacking: topics that must be explored further or that have not received enough attention yet. These topics are probably the ones with the most research potential in the moment.

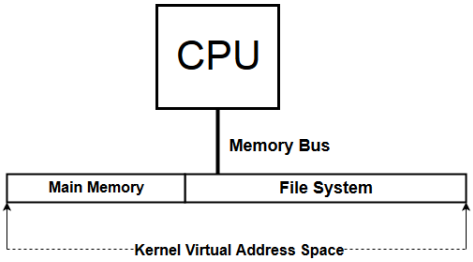

Figure 5.1: Target architecture – single address space model

Table 5.1: Comparison of studied storage solutions.

Problem	NVM System Samples							Status
	TMPFS	DAX	PMFS	BPFS	Mnemosyne	HEAPO	SCMFS	
Consistency Guarantee	NO	YES	YES	YES	YES	YES	YES	**ADVANCED**
Atomicity	NO	YES	YES	YES	YES	YES	YES	**ADVANCED**
Endurance	NO	NO	NO	NO	NO	NO	NO	**ADVANCED**
Access Interface	NO	NO	NO	NO	YES	YES	NO	**LACKING**
Asymmetric Latency	NO	NO	NO	NO	NO	NO	NO	**LACKING**
Metadata Management	NO	NO	YES	YES	YES	YES	YES	**LACKING**
Space Efficiency	NO	NO	NO	NO	YES	NO	NO	**N/C**
Software Overhead	NO	YES	YES	YES	YES	YES	YES	**ADVANCED**
Block/Page Allocation	NO	YES	YES	YES	YES	YES	YES	**ADVANCED**
Energy Efficiency	NO	NO	NO	NO	NO	NO	NO	**N/C**
Garbage Collection	NO	NO	NO	NO	NO	NO	YES	**N/C**
Memory Protection	NO	N/A	YES	YES	N/A	YES	YES	**LACKING**
Cache Consistency	N/A	NO	YES	YES	YES	NO	YES	**ADVANCED**
Cache Optimization	NO	NO	NO	YES	NO	NO	NO	**ADVANCED**
Reliability	YES	YES	YES	NO	NO	NO	NO	**ADVANCED**
Write Amplification	NO	NO	NO	YES	YES	NO	NO	**ADVANCED**
Data Placement	NO	NO	NO	NO	NO	NO	NO	**LACKING**
Access Transparency	YES	YES	YES	YES	YES	NO	YES	**ADVANCED**
Data Duplication	YES	YES	YES	NO	NO	YES	YES	**ADVANCED**
Fragmentation	NO	YES	NO	NO	NO	NO	NO	**N/C**
Persistent Cache	NO	NO	NO	NO	NO	NO	NO	**N/C**
Parallelism	N/A	N/A	N/A	N/A	N/A	N/A	N/A	**N/C**
Mounting Time	NO	NO	NO	NO	NO	NO	YES	**N/C**
Scalability	NO	YES	NO	NO	NO	NO	YES	**LACKING**

5.1 Architecture

Although the studies discussed in this survey work with different architectures and have different technologies as target, it would be extremely hard to point and discuss the directions and

trends in NVM related studies for all of these distinct models. Furthermore, it seems clear that the dominant approach to insert NVM in the current architecture is by connecting it to the memory bus. Thus, Figure 5.1 presents the target architecture. In this approach, memory is exposed directly to the processor and may be accessed directly on byte or word granularity (with load and store commands, for example). From a performance point of view, this is one of the most efficient ways to access persistent memory, but it has a few drawbacks. Additionally, as discussed previously, the presence of multiple layers of cache also impose some challenges.

5.2 Discussion

This section offers an overview about the current level of development, existing solutions and future work on the different topics regarding the area of NVM storage systems. The conclusions presented here are based in the analysis of the mapped studies, current non-academic works in NVM area (like file system and block driver implementations or kernel adaptations) and current market trends. Although many aspects of NVM and many of its applications have been studied throughout this work, the focus of this discussion is on the file system level and its responsibilities. Thus, some contributions like user level programming models and architectures alternatives to the one we established earlier may be not taken into account when discussing the trends and future directions.

Consistency Guarantee and Atomicity

Looking at Figure 4.1, it becomes clear that methods of consistency appears to be the most commonly researched concept regarding NVM file systems. This is no surprise since, in a file system, consistency and atomicity techniques are among the main source of overhead, and when the file system is moved to low latency NVM devices, this overhead becomes even more critical. The result is a large variety of techniques, each with its own focus and advantages. Most of these techniques are strongly based on existing solutions for SSDs and HDDs. Consistency and atomicity techniques designed for SSDs and HDDs usually work on page or block granularity, meaning that, even if an update changes a single word in the file system, a whole block must be updated, causing write amplification. Byte-addressable NVM on the other hand, can work on word or byte granularity, updating only necessary data, and can perform in-place updates, without the need to perform copies. PMFS explores this property by employing a fine-grain journaling technique to log metadata. For user data updates (usually larger) it uses copy-on-write. Another example of adoption of fine-grain logs is presented by the two-level logging technique [44], which logs data at a sub-page level (128 bytes) before merging it with data in the block-based storage. Another potential solution is the log-structured design. Log-structured file systems are very popular solutions for NAND flash storage, since they naturally ensure consistency while also performing wear-leveling. In this model, the file system simply appends data to files (in the form of logs), updating the necessary metadata, and no additional copies are needed. Old versions of these logs can be kept in the file system for long

periods and are erased asynchronously by garbage collectors. However, these file systems are natively block-oriented and additional experimentation would be needed in order to adapt it to a fine-grain solution. As for the atomicity of operations, simple transaction models, like the one presented by Kiln [132] or FSMAC can be adapted. Transactions are useful to establish points of consistency on file system, and they must be very lightweight. Again, the transaction system may need to be modified to deal with small structures, like inodes. It is important to note that, in order to adopt such models, it is necessary for data to be ordered when moved from the cache.

Endurance

The endurance problem is one of the biggest concerns regarding NVM devices. Although the impact of limited endurance of NVM is critical, much research has already been dedicated to this matter (see Figure 4.1). Furthermore, the endurance of these technologies are expected to be greatly improved as they achieve a higher maturity level in their development. Besides, NVM might still coexist with volatile technologies like DRAM and SRAM in many architectures, which also reduces the write frequency in persistent memories and the risk of premature failures. And even so, while such durability is not provided by present day NVM technologies, wear-leveling and other techniques can be decoupled from file systems implementations by, for example, implementing them under the memory management, or by adopting specialized hardware (e.g. remapping logical to physical addresses when necessary, similar to the solutions presented in SSDs FTL). These solutions are transparent to the file system, which helps such systems to be a bit more technology agnostic and adapt to further improvements on NVM.

Access interface

Even though many persistent memory file systems [21] [32] are POSIX compliant and access to these systems are basically provided by traditional functions (like read/write and *mmap*), it does not seem that these are best fit to access memory-like devices. Operating system optimizations (like XIP and direct mappings) as well as application level directives (for example, persistent regions and key-value access) have been proposed to leverage the advantages of byte-addressable NVM, but no definitive framework and model exist at the moment. Furthermore, these solutions often present some expressive trade-offs, like trading portability for simplicity or sharing for security. Many of these solutions are based on today's programming models and patterns and as programming evolves and adapts to NVM-enabled environments, the storage interface and access methods must evolve as well. The adoption of NVM brings storage much closer to the main memory, opening new possibilities for data management and enabling mechanisms and models previously unfeasible. Therefore, it seems clear that this area has much to be explored in both system level and programming level.

Asymmetric Latency

The problem of slow writes may represent a challenge in many applications of NVM. In traditional file systems, most of the time, intensively modified data will be kept in cache and volatile RAM, and are written to secondary memory asynchronously. To minimize the number of writes to NVM, traditional techniques of buffering writes can also be employed in NVM systems, especially in write dominant workloads. These methods, however, may result in data duplication and write amplification and there are many cases in which they may not perform well. The impact of the write latency can also be mitigated through fine-grain writes allowed by the byte-addressable NVM technologies. More complex solutions to these problems could involve additional hardware and specific algorithms. An example is presented in [3] and also mentioned in [87], where writes to persistent memory sacrifice durability in favor of lower latencies. Another possibility is the use of read-before-write techniques in lower levels of the memory management to reduce the impact of writes [77]. These solutions involve more complex mechanisms, require additional hardware tuning, are out of the scope of a file system and will probably not be available in the near future, which keeps this topic open for innovation.

Metadata Management

Metadata dictates how user data will be persisted and retrieved from storage and is intimately related to the file system overall structure and functionality. Its design is, therefore, a critical point in optimizing storage over NVM devices. Upcoming byte-addressable NVM brings a lot of questions to the table, like whether hierarchical or flat name space would be a best fit and whether files are even the best abstraction to work with memory-based storage. Although a fair amount of research has been invested in metadata designs, most of them are, at some level, based on structures developed for disks and flash devices, and many of the aforementioned questions remain with no definitive answer.

Currently, a significant amount of effort and overhead is performed in order to persist data. In-memory data, organized in structures and objects, for example, needs to be transformed into persistent formats, like tables and files, before it can be persisted. Since NVM can be accessed in bytes, memory-like structures can be used to store data. However, it is unclear how this memory could provide the flexibility and portability of regular files and databases, since data and metadata structures may be application or operating system dependent. Additionally, many studies agree that current metadata is optimized for block-based storage devices and propose their own file system metadata models.

Furthermore, NVM byte-addressability and the shorter distance between storage and process memory enables new features, like more meaningful metadata [1] and page table structured files. Many of the points stated in Section 5.2 are valid for the case of metadata as well. New features, as well as metadata structures, should evolve along with data and storage systems concepts and perhaps today's metadata may even gain new purposes in future systems.

Space Efficiency

One of the main characteristics of NVM is that it is more dense than DRAM and it has the potential to surpass flash memory in terms of capacity. In high-end systems and servers NVM is supposed to easily provide tens or hundreds of terabytes of storage space. Thus, NVM can actually represent a solution for today's storage limitations. Besides, thanks to its byte-addressability and low latency, metadata structures may be simplified and its amount reduced giving space to additional user data. Storage capacity may still be a critical factor for embedded and mobile devices, especially since these devices have been target of much research and development in the latest years. In systems such as these, where the space efficiency is mandatory, compression algorithms (some of which are designed for NVM [77] [114]) may be employed to further improve storage utilization.

Software overhead

Most of the software overhead in traditional file system models has been already eliminated by most NVM file systems by simply eliminating layers like I/O schedulers and page caches. Further performance improvement is possible by also bypassing caching procedures and mapping pages directly to user space, using mechanisms like XIP [32] [61] that are already supported by Linux. Besides, much of the complexity presented by some file system policies, like techniques to avoid fragmentation, can be simplified or even discarded by NVM-based file systems. Further improvements may be implemented to avoid more low-level procedures, like entering kernel mode. One way to avoid switching permission levels is through memory mapping, where physical persistent memory pages are mapped into the process address space and no further system calls are needed to read from, or write to, these pages. Memory mapping can also be used to implement more complex and robust models of persistence, like the persistent variables and persistent regions presented by Mnemosyne [118]. A possible alternative is shown in [10], where operating system and file system level permission checks are implemented in hardware, eliminating the need of switching to kernel mode. Finally, as extensively explored in some studies [107] [108], storage systems can also benefit from moving the processing to the application side, bypassing the kernel and its I/O layers that are usually the main source of software overhead.

Block Allocation

Regarding block and page allocation, most policies aim to provide wear-leveling and mitigate fragmentation levels. However, block allocation policies that seek to maximize the device's lifetime may be redundant, since many alternative methods to improve NVM lifetime have been proposed, some of which may be decoupled from the file system itself (see Section 5.2). Furthermore, it is expected that, as it reaches more mature levels, the endurance of NVM become less of a problem. Without file system level implementations of functionalities like wear-leveling and garbage collector (see Section 5.2) the impact of fragmentation in the system's overall performance is significantly reduced. Thus, complex block allocation methods may be unnecessary in many cases and may, in

fact, represent unwanted overhead for the file system. On the other hand, some specific techniques may be employed to boost performance in some cases, like pre-allocation methods presented by SCMFS [126] and Ext4 that simplify the allocation of memory for files and may also be used for consistency purposes (e.g. log structures). These techniques are heavily based on today's workloads characteristics, therefore, as workloads evolve to embrace the characteristics of upcoming NVM technologies, new allocation techniques may be studied to better reflect application's necessities.

Energy Efficiency

Much like space efficiency, energy efficiency may be greatly improved in existing systems by NVM thanks to its capability to hold data without an energy source. This is another factor that allows the adoption of large amounts of NVM in the main memory level partially or completely replacing DRAM. Still, there may be specific cases where energy is scarce and must be used carefully. In these cases, the energy consumption of NVM-based storage may be improved through low-level techniques like copy-before-write [77]. Since writing to the device is the main source of energy cost in NVM (writing to NVM is significantly expensive), generic methods to reduce writes (like improved metadata design, and techniques to minimize write amplification) can also help to create a very low energy storage layer.

Garbage Collection

The garbage collector is usually employed to asynchronously detect and erase data remaining from consistency and wear-leveling mechanisms, especially in solutions for flash memory. Thus, the garbage collection mechanism depends heavily on the consistency method implemented by the file system and whether it implements any wear-leveling method. In some cases, like in log-structured file systems, garbage collection could be used to eliminate older versions of pages or blocks. There are a few simple implementations of garbage collection designed for this purpose, in the literature, that can easily take on this task. In general, the necessity and, occasionally, the implementation of garbage collection will naturally depend on the mechanism generating the data to be collected, which makes the possibility of a general-purpose garbage collection algorithm unlikely.

Memory Protection

The memory protection problem exposes the possible hazards of persistent data being in the same memory hierarchy level of volatile data. Such hazards may expose critical data to bugs or even be explored by malicious attacks (these are out of the scope of this topic). But even if traditional block-based storage isolation level is desirable in some cases, it could also mean increasing the distance between persistent memory and volatile memory as well as CPU, which might result in sacrificing performance and adding complexity. Currently, the trend on protecting byte-addressable NVM exposed directly to the CPU seems to be exploring the protection bits of page table entries. Although, on today's hardware updating the protection of pages is a relatively expensive operation,

more complex solutions, like memory protection keys [24] are already being developed. However these solutions are still in experimental phase and it is unclear whether they are robust enough to be considered a definitive solution and what are their impact on a real NVM-enabled system. It is also worth to mention that some of these solutions depend on new hardware, a factor that, in some cases, may cause some resistance in the adoption of these solutions in the near future.

Cache Consistency

In order to ensure the persistence and integrity of data in a NVM file system, having a processor cache that is also persistent would probably be the ideal solution. However, the access speed and endurance of current NVM technology makes a NVM-based cache unfeasible. Even if the endurance of these technologies are expected to be greatly improved in the future and will not be a problem for main memory, it is not clear whether this will be true for the cache level as well, since the write intensity is much higher in this layer. Besides, even the most advanced (in terms of development) persistent memory technologies at the moment are significantly slower than the SRAM (Static RAM) used in most caches. Therefore, persistent caches will probably not be a reality in the near future, and alternative ways to deal with cache consistency are required.

In this case, using memory fence and flush instructions, together to ensure that writes to the file system reach the persistent memory, seems to be a reasonable solution that works for most cases and applications. However flushing cache lines may be an expensive process and, in some cases, frequently performing cache flushes may dramatically degrade performance. In fact, a study that presents a comparison of different cache modes [6] shows that in some cases, using a combination of memory fence and flush along with write-back cache mode performs 2 times worse than write-through and, in some scenarios, it may be worse than not caching data at all.

To enforce consistency with volatile cache, new processor instructions have been developed in the last years, to improve on the ordering and flushing scheme, by, for example, not invalidating the cache lines on flush. Naturally, adopting such instructions may require some adaptations either in application or operating system level or both. Although it seems that, at the moment, using these instructions is the trending way to ensure cache consistency, new mechanisms for more robust and transparent cache management (like, for example, the epoch barriers [21]) may be researched in the future providing alternative approaches to this matter.

Cache Optimization

Cache and buffer optimizations are usually designed to improve SSD-based storages. Hence, most of these optimizations do not apply to byte-addressable NVM-based file systems. In an architecture where long-term storage is combined with main memory, the processor cache is the main point for cache optimization. Thus, techniques that try to enforce consistency between cache and NVM efficiently (see Section 5.2) represent the main contribution. Another possible point of improvement is the TLB. For instance, SCMFS [126] proposes the use of superpages (2MB pages)

to reduce the impact of in-memory file system in the address translation performance, by allowing larger regions of memory to be mapped by a small number of addresses. This is also a good example of a method to address possible problems with scalability, which is a relevant field that will probably receive significant attention in the next years.

Reliability

Reliability in NVM systems may be achieved in different levels. For most cases, ECC [38] seems to be a reasonable solution that is already in use in today's memory and storage devices. This can avoid data corruption caused mostly by unexpected physical problems that would be, otherwise, made persistent in NVM devices. NVM also offers the possibility to improve on another common reliability mechanism, known as check-pointing. Check-pointing can be used to provide fault tolerance, security and instant process restarts. Besides the existing check-pointing mechanisms, a few models have already been proposed for NVM-based storage [113] [91]. Other common fault tolerance methods based on data redundancy, like disk RAIDs, can also be useful to leverage the security of NVM file systems. It is unclear, however, how this data redundancy could be treated in a NVM-based architecture or the impacts of such techniques and none of the studies analyzed in this survey deeply explored this possibility.

Write Amplification

Along with additional CPU cycles caused by software overhead, unnecessary writes to the backing storage is one of the main sources of performance degradation in NVM storages. The write amplification problem is closely related to other functionalities like consistency and reliability mechanisms, wear-leveling algorithms and atomicity methods. Although much effort has been put in these mechanics, only a fraction of these studies focus on presenting methods to reduce write amplification. Techniques like fine-grain logs, direct mappings and atomic in-place updates contribute to reducing overall write amplification, but other techniques commonly responsible for write amplification (like copy-on-write and tree-based updates) are still very common among the studied solutions.

Data Placement

In the target architecture, adopting a block placement policy may be very useful in cases where a hybrid memory approach is adopted (e.g. DRAM and PCM). Although copying data in memory is certainly undesirable, exploring faster memories (like DRAM) for caching frequently used data (like metadata) or as write buffer may be interesting in some cases. Thus, some policies are needed to determine whether moving data from a device to another is advantageous and whether using fast volatile memory to improve performance when accessing persistent data is acceptable. Although some studies explore the block placement topic, only a few are actually designed for hybrid memory layer model similar to the target architecture described in Section 5.1.

Access transparency

Regarding byte-addressable NVM, most studies try to leverage NVM characteristics in order to reach maximum performance and, in many cases, avoiding traditional block-oriented approaches and layers like block drivers. In this context, transparency is related to compatibility and legacy systems accessing NVM storage in an efficient way with traditional interfaces. For future systems, where NVM is accessed like regular DRAM, it should not be one of file system's concerns to ensure compatibility with technologies, as the backing device hardware should be (it could be interesting, however, in case of hybrid memories).

Data Duplication

In an architecture where the file system is built in the main memory, the amount of data replication will naturally be reduced, since, in this case, many of the buffers, caches and software layers that compose the traditional storage system will no longer exist. Also, since file data is now in a byte-addressable memory and may be accessed by the CPU directly, processes may access persistent data directly without the need of copies or page swaps. Thus, a mechanism like XIP that allows pages to be directly mapped into user space represents a solution that involves little to no data duplication at all. Of course, since data is directly written to persistent memory without passing through the file system API, regular consistency mechanisms, like logging, would not be able to provide the same level of reliability. To improve security in these cases, either some kind of consistency mechanism must be implemented in application level, or a more robust (and probably more complex) solution would have to be proposed at the operating system level.

Fragmentation

Fragmentation in file systems has always been of big concern: in HDD fragmentation could result in long seek times caused by the magnetic disks mechanical parts, while in NAND flash SSDs fragmentation could negatively impact the performance of block merges and garbage collection. Byte-addressable NVM on the other hand does not present many of these characteristics and external fragmentation is only a problem in specific cases, like, for example, when files are always allocated contiguously in virtual memory [126]. Also, due to the fact that NVM is expected to be adopted in large amounts in the future (e.g. petabytes in the same address space), the size of memory units, like pages, may need to be adapted in order to efficiently address such a huge memory. Adopting larger page sizes (that today range usually from 4 KB to 16 KB) could lead to internal fragmentation. This is a problem connected to the topic of scalability on new architectures, which is an area that will probably need more attention and intense research in the near future

Persistent cache

In most cases, persistent cache acts as disk caches or write buffers, and do not quite fit in the architecture targeted by this discussion. Most policies designed for persistent caches and buffers usually aim to serve a very slow I/O storage device in the lower memory layer, like NAND-based SSDs or HDDs. On the other hand, as discussed in Section 5.2, making processor cache persistent may be a robust and, perhaps a definitive solution to the volatile cache and reordering problem. However, due to latencies and endurance of current NVM technologies, it seems unlikely that such non-volatile caches will become a reality in the near future, or, at least, that they will not replace the existing volatile cache layer. Nonetheless, these studies do help to illustrate that, if the non-volatile processor cache approach is adopted in the future, existing cache policies may not be the best fit for such model.

Mounting time and parallelism

Mounting a file system sometimes may be a problem in flash-based SSDs, since it involves locating and copying metadata from the storage to main memory and metadata location may not be fixed. In byte-addressable NVM, however, metadata may be easily moved between memories or even accessed directly in NVM. Although unifying operating system and file system metadata is not a trivial task (for example, Linux has its own representations of inodes and super-blocks and storing these could limit portability) this concern seems to be much more related to the metadata management issue. As for parallelism, in the scope of this SMS, this term refers to exploration of hardware internal parallelism that is related to SSD architectures and is not relevant for the established architecture (Section 5.1) and scope.

Scalability

As the amount of memory available in the memory bus increases, some challenges may emerge. These challenges are generally connected to the concept of a large amount of memory in a single address space. When NVM is attached to the memory bus, it will, at least initially, use memory management resources, like memory controller, TLB, translation mechanisms and the bus itself. It is unclear how these mechanisms will perform with large pools (petabytes) of memory on the same bus. Compared to consistency and endurance problems, scalability issues are not very frequently addressed in academic studies. This may be caused by the fact that NVM technology currently has limited capacity and is quite expensive. However, as the technology reaches its maturity, computers with large amount of main memory will be available, especially for servers, and today's operating system might not be ready for such a huge memory in a single flat address space. Thus, this topic is very important and, although research on it seems to be lacking, it will certainly receive more attention in the future.

One instance of the scalability problem is discussed in SCMFS [100]: the difficulty of efficiently addressing a large dataset. It points out that addressing the whole memory (in this case

it also includes the file system) using 4 KB pages may become problematic, since it would have a great impact on TLB. In this case, the adopted solution was to include an alternative type of page, called super-page, that would be 2 MB long. As mentioned previously, this approach has some drawbacks, like internal fragmentation, coarse-grain memory protection and potentially write amplification issues.

On the other hand, the VFS design can be inherently cache and TLB inefficient, which makes in-memory file systems challenging to properly scale [56]. Hence, pVM adopts the NUMA node abstraction to work with NVM, making allocation and data placement more flexible and scalable. pVM solution is based on virtual memory, persistent memory regions and object store, and is closely related to solutions proposed by Mnemosyne [118] and HEAPO [43] than to a file system like SCMFS. This solution is not designed to replace file systems, however, as it does not provide file system functionality or semantics (e.g. file and directory hierarchy and user permissions) and it also has some portability limitations.

5.3 Industry Status and Trends

This section is dedicated to recent topics, trends and solutions discussed in the computer industry regarding the adoption of existing and upcoming NVM technologies. We start by understanding the industry needs that drive NVM adoption. The amount of data stored and exchanged by today's applications has been growing at an unprecedent rate, a growth that is not expected to slow down in the coming years. The huge latency gap between volatile main memory and persistent storage imposes a big challenge on systems design. Even with the growing sophistication and adoption of SSDs, the distance between volatile and persistent data still one of the main bottlenecks in current architecture, as applications demand faster access to persistent data. That leads to the multiplication of hardware and software tiers, which in turn lead to more data copying and movement, reducing overall system efficiency.

In order to make the memory/storage stack more efficient, it becomes necessary to consolidate the different stack layers, reducing data copies and movement. This can be achieved by new technologies introducing characteristics, such as low latency, high density, low cost/bit, high scalability, reliability and endurance. Despite the fact that no memory technology today can provide all of these features [99], new NVM technologies being developed promise to narrow the gap between memory and storage. Driven by this perspective, significant effort is being directed to create standards, interfaces, functions and programming models dedicated to allow efficient adoption and usage of NVM by operating systems, programming languages, and applications.

5.3.1 Tools and Standards

The ACPI (Advanced Configuration and Power Interface) specification [115] is a standard used to allow operating systems to configure, manage and discover hardware components. Version 6.0 added the NVDIMM Firmware Interface Table (NFIT) to the standard in order to provide information about features, configuration and addresses of NVM devices. The NFIT describes persistent memory regions, provides hints to make efficient use of cache flushes necessary to ensure durability of writes operations to these regions, and the definition of block data window regions in case apertures are required to access the NVM. The JEDEC Byte Addressable Energy Backed Interface standard [47] specifies the low-level access interface for NVDIMM devices. It is intended to simplify BIOS and NVM access and to provide a single interface (that may have multiple implementations) to the operating system.

The Linux PMEM driver is a block driver based on the Block RAM Driver (also known as *brd* used to create RAM disks) designed to work with NVMDIMM [133]. PMEM was developed by Intel and eventually incorporated into Linux kernel 4.1. PMEM uses a memory addressing range reserved by the system, similarly to ranges used to communicate with I/O devices. PMEM may be used to create and mount regular file systems over memory regions, whether the memory is persistent or not, allowing users to emulate persistent memory with PMEM. Currently, the PMEM driver is being updated to support the features of ACPI NFIT.

The Linux LIBNVDIMM is composed of the NFIT-aware PMEM along with a few other drivers currently under development. It is a kernel subsystem adding NFIT support to Linux [122]. One of the main capabilities implemented by these drivers is the support for the block window transfer feature described in the ACPI specification. The block window transfer is more complex than regular direct access to NVM as it requires an additional translation steps for the CPU to access physical addresses in the persistent memory device. However, by adding this additional layer of address translation, it allows access to memory not mapped in the kernel address space, which means that the amount of available memory is not limited by the OS addressing capacity. It also means that memory accessed using Block Windows are not in danger of being affected by stray writes, ensuring memory protection.

Another feature provided by PMEM is the block mode access to NVM. As the name suggests, in block mode data is transferred to persistent memory in blocks, in a similar fashion to block drivers. The block-grained access, although inherently slower than load/store based access, has a few key advantages, for example, when it comes to handling memory errors, which are more difficult to address when executing direct access to NVM. Accessing NVM in block mode allows errors during NVM access (e.g. "bad blocks") to be treated by the kernel, while managing errors when accessing NVM directly with load/store instructions is much harder and might cause a system crash. The block mode access also ensures atomicity at block granularity, which may be useful in some cases.

Another Linux feature implemented to improve its compatibility with NVM is the DAX (Direct Access) functionality [23]. The concept behind DAX is very similar to the concept of XIP (eXecute In Place), employed by some of the file systems discussed previously [32] [92]. The principle of DAX is to bypass Linux page cache, avoiding additional copies of data that, when storage is built over NVM, would only represent unnecessary overhead. With DAX NVM can also be directly accessed by applications through the mapping of memory regions in the address space of user processes. In order to support DAX, file systems and block drivers must implement a few functions that compose the DAX interface, allowing the kernel to perform specific operations, such as allocating pages using page frame numbers. To date, the file systems that offer DAX support are Ext2, Ext4 and XFS. DAX combines improved NVM access mechanisms with modern and mature file systems designs.

5.3.2 Architecture Support and Limitations

Even though NVM presents highly desirable attributes, like low latency and high density, architectural support for these memories is still missing. A good examples of this fact is regarding the limited physical addressing capacity of current machines. An address space containing terabytes (or even petabytes) of NVM is well beyond what today's processors are capable of supporting [33]. Additional address bits would be necessary to work with large-scale memory (although there are some workarounds for it, such as mapping NVM regions temporarily into the address space of CPUs). However, it is still unclear the overall implications of scaling memory to that amount (increased occurrence of TLB and cache misses, increased overhead of memory zeroing and copying large amounts of memory, etc.). Ultimately, memory scaling is an open challenge that processors designs must cope with.

The processor caches have issues with persistent memory as well, some of which have already been discussed in Section 5.2. As explained in Section 4.3, the fact that writes to NVM are retained in volatile cache and may be reordered before reaching persistent memory represent a challenge on ensuring persistence and consistency. Existing barrier and cache flush instructions, although useful to mitigate these limitations, represent a performance drawback as they are expensive operations and may serialize execution within the processor pipeline. When flushing a cache line with these instructions, there is usually no guarantee that data is immediately written back to NVM. Hence, Intel introduced a few instructions to their new processors, like *clwb*, *pcommit* and *clflushopt*. These instructions are similar to the *clflush* and *mfence* instructions described earlier, except that they do not invalidate cache lines, or stall the CPU and, in the case of *clflushopt*, may be pipelined. Additionally, the *pcommit* instruction can be used to ensure that writes accepted by the memory controller are committed to the persistent memory synchronously.

Another example of processor support for NVM is the addition of memory protection keys (see Section 5.2). This mechanism added new registers to the processor that allow systems to define up to 16 protection keys, assign these keys to page addresses and choose which of these

keys are writable and which are read-only. These keys provide an efficient method to lock a range of memory against writes while also avoiding changes in page tables and TLB flushes. The TLB is yet another point of improvement in current architecture. This address translation buffer is critical to address translation (and, therefore, to the whole system's performance) but may represent a scalability limitation on systems with huge amounts of memory. Additionally, some memory subsystem operations require flushing this buffer which, like in the processor cache example, is a very expensive process. Even though software techniques, such as segmentation and coarse-grained pages, may be used to alleviate the scalability problem, any improvements to this mechanism may be an important step towards a more efficient NVM addressing scheme.

5.3.3 Programming Models

As briefly discussed in Section 5.2, traditional access methods and programming models may not be the best fit for NVM storage due to their singular characteristics. Therefore, different programming models and access methods aiming to explore the features of byte-addressable NVM have been introduced [118] [25]. The Storage and Networking Industry Association (SNIA) has defined and published the NVM Programming Model (NPD) specification in order to provide some directions for developers to provide common and extensible NVM access model. The specification is also useful for users to understand what can be expected and what operations can be performed over NVM systems. The NPD defines multiple modes of access (e.g. file mode, block mode), what they have in common, how they differ, at what level of the architecture they operate and what kind of operations and attributes should be supported by these modes. The specification provides detailed information about methods to discover the supported operations provided by a specific implementation and the high-level description of these operations (inputs, behavior, outputs, etc.). Finally, NPD also provides a few use cases to illustrate the usage of the specified behaviors and describes a few directions to make programming interfaces compliant with the specification. The NVM Library (NVML) [25] is a set of open libraries designed to provide applications with easy and efficient access to NVM. The NVML follows the design principles that are specified in the NPD, but also adds an array of specific features to make development for memory storage more intuitive. It has tools to work with different abstractions such as objects, files, append logs and blocks. It also exposes to users useful low-level functions, like cache flushing, optimized memory copy and optimized file mapping. On higher-level libraries, NVML allows atomic transactions, persistent pointers and lists as well as synchronization for multithreading. Finally, NVML also provides a C++ version of the API (still under development), allowing more intuitive and robust object-oriented programming over NVM.

6. ATOMIC ACCESS TO MEMORY MAPPED FILES

One of the most important discussion topics identified in Chapter 4 refers to the interface with which NVM-based storage systems are accessed: what are the most efficient ways to access data in NVM? How should be NVM be exposed to applications? When are file systems semantics really necessary and when are alternative models (like key-value stores) more appropriate? As we have seen, these questions have driven much of the research in the NVM area [97][4][43][35].

At the center of the NVM access topic is the traditional *mmap* system call. Memory mapped files are well-known and efficient methods to give applications memory-like access to persistent files' data. Furthermore, with the adoption of the eXecute-In-Place (XIP) concept (already integrated in Linux) [117], it is possible to simply map files from the file system into the application's address space. Due to the freedom it gives, this method is used by many libraries and frameworks designed to provide more dynamic and efficient storage models such as persistent regions and key-value stores [118][20][43][56].

In this chapter, we present two variations of the XIP-enabled file mapping that add transactional semantics to the *mmap* function, allowing applications to prevent file corruption. For that, we modify the Linux kernel to perform copies of the file data being update by the application, ensuring that the original content of the file may be retrieved at any time. After that, user data are made durable once the file is synchronized (through a *msync* call for example). After the file synchronization is completed, the new file version is guaranteed to be in persistent state.

It should be made clear that both these mechanisms are designed for an architecture where persistent memory is attached to the memory bus. Naturally, this memory must be byte-addressable in order to be accessed by the CPU. Therefore, for this chapter, we refer to byte-addressable NVM simply as NVM for simplicity's sake.

6.1 Motivation

File mapping is a flexible and efficient alternative to traditional read and write based file access. It allows users to access a file like a memory array by simply mapping a region of the process address space to a file and copying the file's blocks into page cache on demand. In XIP-enabled file systems, this method is even more interesting, since applications can map their virtual addresses to NVM physical memory directly, thus avoiding, among other things, the page cache copy step.

The price for this simplicity is the loose file integrity guarantees. In traditional mapping, pages of a mapped file may be made persistent by the OS at any time, creating a relatively large critical window of time where a system crash could damage the integrity of the mapped file in storage. In XIP mappings (which we will refer to as direct mappings from now on), integrity is even more fragile, as writes to the mapped file are expected to become persistent (although they are usually cached in the processor) immediately and at byte granularity.

As result, in most cases applications and libraries based on file mapping have to implement their own consistency models (e.g. based on transactions). However, implementing such mechanisms may be tricky and loss of data caused by bugs in the code may be fatal. Furthermore, developing such low level mechanisms or employing third party code to ensure data consistence may represent a penalty in optimization and portability in some cases. Therefore we believe that the operating system should provide the means for users to ensure their data is safe (in terms of consistency) at all times using file mappings. It is important to note that more fine-grained and specialized solutions, like the ones implemented by NV-HEAPS and Mnemosyne [118][20] may be more efficient and optimized in many cases. We argue, though, that a solution based on existing mapping implementation in the OS level has the advantage of being more flexible, portable and simple while also being able to maintain a common traditional file system interface.

As NVM technologies grow in maturity and popularity, new implementations of frameworks and tools designed for NVM-based storage, like persistent heaps, will become more common. Since file mappings is a simple and straight forward manner to building persistent structures, like objects or data tables, over NVM, it is fair to assume that this may be a popular design trend for these pieces of software. Considering that most of these tools will also have to offer some level of data integrity to their users, adding consistency guarantees to the semantic of functions like *mmap* could significantly reduce the amount of responsibility given to third party systems and libraries and a save a lot of work for both tools developers and end users.

6.2 Related Work

Guided by the current NVM promises of fast access in byte granularity and non-volatility, many researchers have proposed alternative ways to access NVM to use for NVM-enabled systems instead of traditional file system semantics. Such models usually aim to give users the possibility to manipulate persistent data in a format closer to that of memory structures, hence avoiding the process of serializing it into file or database formats. These solutions also seek common goals as well such as minimizing processing overhead and write amplification. Many of these tools, however, depend on an underlying file system and usually rely on directly mapped files to freely manipulate data in NVM.

One such framework is Mnemosyne [118]. Mnemosyne provides programming friendly mechanisms to handle persistent data, such as persistent variables and persistent regions. Mnemosyne uses mapped files in NVM as back store and manages the access to these areas. In order to ensure persistence and consistency, Mnemosyne implements and offers a variety of options, such as append and shadow updates as well as atomic transactions. It also offers a few primitives to make the job of creating custom consistency mechanisms easier.

Another tool designed to bring NVM closer to application level data structures is NV-Heaps [20]. NV-Heaps is based on the concept of persistent objects and heaps and it aims to

minimize bugs and reliability issues when working with NVM while also exploring the performance advantages offered by them. It provides an array of primitives and abstractions that allow users to build persistent objects while also allowing some additional useful operations such as definition of atomic blocks and persistent pointers. The actual heaps are stored on directly mapped files and managed by the NV-Heaps user-space code. For atomicity, NV-Heaps implements fine-grained and log-based transactions that perform security copies of persistent objects before allowing users to modify them.

Due to the demand for stronger consistency constraints on mapped files, a few studies have proposed methods to increase these mechanisms' reliability and allow atomic updates on mapped files. One such study is described in [95], and proposes a failure atomic variation of the *mmap* and *msync* system calls. It presents a very simple concept: data written to a mapped file must be written to disk only during the *msync* call and such write must be made atomic (in this case using a redo journal). Failure-atomic *msync* is designed and implemented for (and evaluated on) I/O storage devices such as HDD and SSD and thus does not take in consideration NVM characteristics. Similarly, the NOVA file system [128] provides an atomic *mmap* mechanism to provide the means of ensuring user data integrity. NOVA's solution is a little more complicated as it involves creating and mapping replica pages (copies of the mapped file's pages) on NVM and copying data from this replicas back to the original data page during *msync*.

Table 6.1: Summary of the characteristics of the file mapping mechanisms in our version of PMFS

	MAP_SHARED	MAP_COW	MAP_ATOMIC
Guarantee of User Data Consistency	NO	YES	YES
Process writes to	NVM	DRAM	NVM
Perform Out-Of-Place Updates	NO	YES	NO
Shareability	SHARED	PRIVATE	SHARED

6.3 Design of the Solution

In this work we present two different solutions to the problem of consistency on mapped files. The first, called Copy-On-Write mapping, creates copies of updated file data while exploring the superior performance of DRAM to mitigate performance loss. During writes to a mapped file, the mechanism performs copies of the original file pages to new pages allocated in the system's volatile RAM, hence the name Copy-On-Write (COW) mapping. The second one, Atomic mapping, is based on performing back up copies of updated pages and storing them in the file system to perform file recovery after crashes or kernel panics. This sections describes in detail both solutions as well as their advantages and drawbacks. Table 6.1 shows a summary of the characteristic of all the mapping methods studied in this throughout this work.

6.3.1 Copy-On-Write Mappings

The concept behind COW mappings is quite simple: in order to protect the original file, the operating system creates a copy of the file block being updated and write updates to the copy. The copies are stored in main memory, which in our target architecture is composed of DRAM. We choose to keep the copied pages in volatile memory in order to improve the mapping's performance on multiple overlaying writes while also compensating for the additional overhead of copying pages from NVM. This is because writing to volatile RAM is considerably cheaper than writing to NVM (although the size of this gap varies depending on the NVM technology). As we will show later in this work, storing these copies in NVM would be much more expensive, specially in more write intensive workload. Figure 6.1 illustrates how a file mapped with by this method would look like in the memory hierarchy.

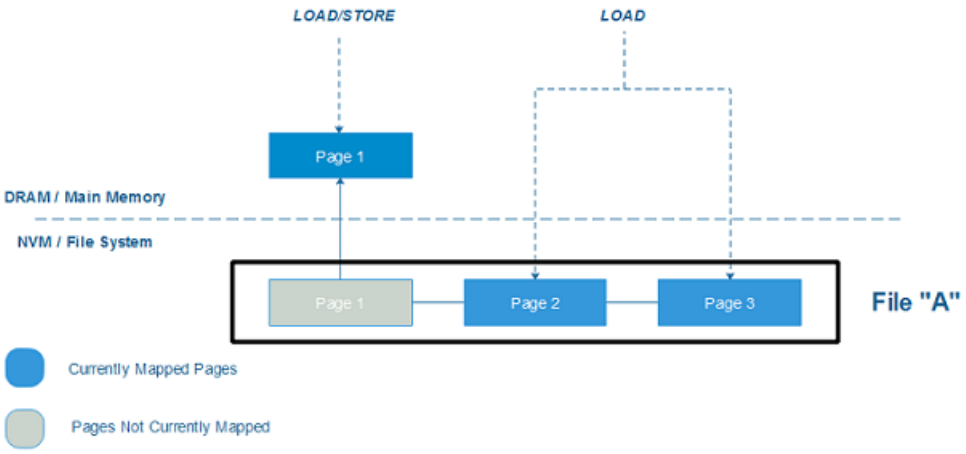

Figure 6.1: Copy-On-Write mapping scheme: data may be read directly from NVM but must be copied to DRAM in order to be updated.

On the other hand, reading from the COW mapping does not trigger the copy-on-write process at all. When being read a COW mapping works similarly to the regular direct mapping: the file's data blocks are accessed directly like any other memory page. This may greatly improve COW mapping efficiency the system perform copy-on-write only on the necessary data blocks. Also,in this scenario this approach does not necessarily represent a performance hazard, since read operations are significantly faster than writes operations in NVM and closer to DRAM performance [97].

When the mapping is synchronized with the original file (through the *msync* call), these updated copies are atomically written back to the file. After the synchronization is performed, the file is in a new consistent state. If during the writing or the synchronization the system halts for some reason (energy outage, hardware error, kernel panic, etc) the original non-modified version of the file may be easily recovered. Additional reads and writes to the faulted segments of the mapped

file are still served by the copied pages: no additional kernel interrupts are needed in this case. The copied pages are only discarded once the file is unmapped.

With this approach, the user process never writes directly to the mapped file, but instead to copies of the file's blocks. The user is responsible for defining when the mapping must be persisted by explicitly issuing a flush operation, hence deciding whether the file is in a consistent state. Therefore, COW mappings ensure file consistency at all times, protecting data against failures. The main disadvantage of this solution is naturally the additional overhead from copying pages back and forth from the file system, which, makes COW mapping significantly slower than regular direct mappings even when employing DRAM as a buffer. However, COW mapping is a very flexible mechanic, and its performance highly depends on the application's needs and on how it manipulates the mapping. Our results show that in some cases, intensively writing to a COW mapping is considerably faster than writing directly to NVM through direct mappings.

6.3.2 Atomic Mappings

Our second proposed mechanism, atomic mapping, is a variation of the copy-on-write mapping discussed earlier. Just like COW mappings, data is read directly from NVM by mapping the file's blocks as pages in user process level. However, during writes to the file's blocks, instead of coying the blocks' contents to main memory, security copies of these blocks are stored in the file system itself. These security copies are never actually accessed by the user process: they are kept in separated blocks of the file system allocated specifically for this purpose. These copies are kept for recovery purpose only, being used to overwrite the file's blocks modified by the user-level process in case of crashes. The user process instead writes to the original file block directly. The design of a file mapped by this method is shown in figure 6.2.

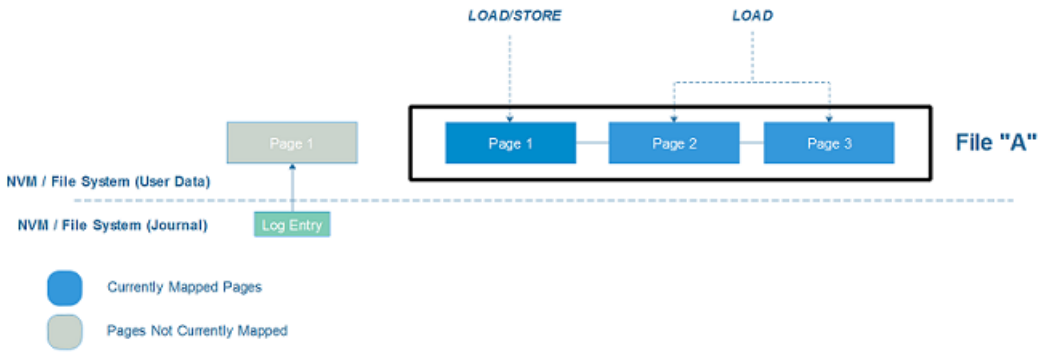

Figure 6.2: Atomic mapping scheme: file data is accessed directly from NVM and security copies are stored in the file system and logged into the journal.

The security copies are kept by the file system until explicitly released, through the mapping synchronization process (implemented by the *msync* call). In this case, the original file is not touched by the mapping synchronization process, therefore there is no risk of data corruption. If the file is

unmapped or the process halts (either unexpectedly or as a valid result of the process execution) before the mapping is synchronized, all non-synchronized blocks are replaced by their respective security copies, and all modifications made to them are lost. This property not only ensures that the file will keep its integrity whether the writes to the atomic mapping are successful or not but also gives the user more control over the file state and the data flow.

After the security copies have been released by the file system, new writes to the original file blocks will trigger new copies. This is the main source of overhead and the main disadvantage of this method. The atomic mapping is designed as an alternative to COW mapping and targeted at systems where there is no hybrid memory. In these cases, there is no performance advantage in copying the data blocks to main memory: the costs of accessing memory pages and file blocks are the same. It is in this scenario that atomic mappings may be a good fit as its process is even more simple than COW mappings and it involves less copies. We evaluate the performance differences of atomic mappings and copy-on-write mappings in both architectures later in this study.

6.4 Implementation

To evaluate the feasibility of our solution, we implemented a prototype of both atomic mappings and copy-on-write mappings over PMFS. As we presented earlier in this work, PMFS is designed for memory-bus attached NVM and with NVM characteristics in mind. It also implements the XIP functionality which is essential for both atomic and COW mappings to work properly. Naturally, our implementation of the proposed mappings use the direct mapping implementation of PMFS as their base, making development easier and more reliable.

We chose PMFS as our test platform because of both its simplicity and flexibility as well as its generality. PMFS is built over an allocated region of the system's memory that is no mapped by the operating system. PMFS manages its space autonomously and writes to memory directly without the need of additional layers, like block drivers. It also keeps the implementation of its mechanisms (such as journaling, transactions and file mappings) quite simple, in order to minimize the processor overhead. Finally, PMFS metadata structure and operations are very similar to the other file systems, both NVM and HDD-based. This is an important factor, since we are more likely to be able to replicate the behavior observed on PMFS on other similar systems.

We also modify the Linux kernel code that is bundled with PMFS implementation. Although the kernel is already tailored to work with PMFS, modifications are necessary to perform actions during page faults, like the copy-on-write step of the COW mapping. We focus on working over PMFS code, and keep our changes to the kernel to a minimum in order to maintain the file mapping mechanism simple and easy to migrate to other systems or kernel versions. As we discuss later in this work, additional modifications to the kernel could improve our solution's performance and usability, however we leave these questions for future works.

6.4.1 Copy-on-write Mapping

In order to provide access to the copy-on-write mapping implementation, we introduced a new *mmap* flag, MAP_XIP_COW. Mostly, all this flag does is mark the mapping's virtual memory area (VMA) as a copy-on-write VMA. It also sets the mapping protection as read-only while marking the VMA as writable, so writing to its pages will trigger page faults. With this configuration, the operating system is able to identify that the VMA is supposed to perform the copy-on-write precedure on page faults caused by protection restrictions. The process of copy-on-write in our implementation is very similar to the existing used for private mappings (MAP_PRIVATE).

After finishing the *mmap* routine, the COW mapping will be ready to be accessed. Naturally, however, since the pages are not actually mapped in the process address space yet (they have not received a virtual address), accessing the mapping address range results in a page fault. The kernel notifies PMFS that a page within a PMFS file mapping is being requested. PMFS then finds the file block being requested and creates a new page table entry based on the block's page frame number. The page table entry is created with the protection defined on the mapping's VMA structure, therefore the entry is marked read-only. Finally, the entry is added to the process page table. This process is enough to provide reading access to the user process: we do not need to perform a page copy at this point.

Once the file's block has been mapped into the process page table, it is ready to be accessed for reading. Writing to it, however, will trigger another page fault, since it is write protected at this point. This time, the kernel allocates a new page from virtual memory and copies the content of the faulted read-only page to this new page. The kernel then adds this copy to the process address space, overwriting the page table entry pointing to the file's data block with a new one, pointing to the copy. The copy's page table entry is made writable, thus allowing write operations over the copy page. The copy page is not cached by Linux (into page cache) and is never written back to the original file, except during the *msync* call. Once the copy page is mapped into the process virtual address space, any further access to its address range will be served by the data in virtual memory instead of the actual file data. This copy will be kept in memory even after mapping synchronization, being only released when the mapping it belongs to is unmapped. Releasing a COW mapping will cause the kernel to release its volatile copied pages back to the virtual memory and any data on dirty pages at this moment will be lost.

During the *(*msync) system call, the kernel checks that the mapping is, in fact, a COW mapping, and delivers it to PMFS to be handled. The first thing PMFS do is create a transaction and add the current inode state to the jornal. PMFS uses the concept of transaction to control atomic updates and uses an undo journal (stored in NVM) to ensure metadata consistency during updates. Next, PMFS checks every page of the address range being flushed, searching for dirty pages. For every dirty page found, PMFS replaces its file block counterpart with a newly allocated block (from persistente memory). The index of the original data block is journaled before the inode is updated, as part of the file system copy-on-write mechanism to ensure file data consistency. PMFS

only journals metadata, such as inodes and inode indexes: file data is never actually written to the journal. Instead PMFS keeps the original state of the inode index (which is a 64-bit address) in the jornal, so as long as we do not modify the original file data block, its data is safe in NVM. If the *msync* process fails for any reason, the file may be reversed to its original state, by simply restoring the inode and the original data blocks' indexes.

After securing the old block, PMFS finally writes the content of the dirty volatile page inside the new empty block and marks the page as clean once again. In order to ensure persistence and avoid part of the data from being retained in cache, we bypass the caching procedure by using non-temporal stores. This may also helps us avoid polluting the cache in some cases: in COW mappings, the data blocks in NVM are not expected to be accessed again anytime soon. The pages in DRAM are the ones that may be frequently needed by the process and the ones we actually want to cache.

After all pages were successfully written back to the original file in NVM, the PMFs transaction is finally commited. By commiting the transaction, one last entry is written to the journal, a commit entry, signalizing that the flush procedure was completed with success. It is also during the commit fase that the old file blocks are released to PMFS. The blocks are delivered to PMFS and handled by a block cleaner thread, that mark them as free blocks once again.

Finally, the life cycle of a mapped file in COW mapping ends when the file is unmapped. This may be done by the user explicitly by calling the *munmap* function or whenever the user process ends its execution. Once the file is unmapped all pages of the COW mapping are released back to the kernel. All non-flushed data in these pages is lost and mapping the file again will require faulting all its blocks back into DRAM. This characteristic may also allow applications to "rollback" modifications to a file in some situations.

6.4.2 Atomic Mapping

To provide access to atomic mapping, yet another option was added to the *mmap* function: the MAP_ATOMIC flag. Its purpose is very similar to that of MAP_XIP_COW, as it prepares the VMA's flags and protection scheme. During the mapping procedure, PMFS starts a new transaction. This transaction will be used to manage the atomic mapping, including the journaled data and the allocation of the security copies.

Just like COW mappings, accessing a page for the first time in an atomic mapping will cause a page fault in order to allow PMFS to map the appropriate file data block into the process address space. The block is mapped as read-only, thus triggering a page fault when accessed for writes. Once it is verified that the mapping is in fact atomic and the page is allowed to become writable, the kernel notifies PMFS that the page table entry related to the faulting address is about to become writable. It is at this moment that PMFS allocates a new block on NVM and copies into

it the content of the faulting page. This copy is kept safe in the journal space of PMFS, so it may be used to restore the file whenever necessary.

These copies are kept along with the transaction until the mapping is either flushed by *msync* or unmapped. At *msync* time all PMFS does is release and commit the transaction referring to that mapping along with all block copies created up to that point. Once both transaction and security copies have been freed, a new transaction is created in order to keep track of the copies generated by the upcoming updates to the file. With this step we ensure that there is always a transaction linked to the atomic mapping. At this point, the file is durable and in a new consistent state. The entries in the process page table are made read-only again. This step is necessary for PMFS to be notified by the kernel whenever these pages are being written. This means that writing to the mapping will once again cause page faults.

Similarly to the COW mapping, when an atomic mapping is unmapped, all the updates not yet flushed to PMFS are lost and the file is reverted to its previous state. However, unlike COW mappings, both versions of data (the original and the partially updated) are kept in NVM and are persistent. This could enable more complex data recovery methods, such as retrieving a transaction from the point it stopped or merging old data with new data to leave the file in a mixed but consistent state. Such mechanisms would not be possible in COW mappings and direct mappings since either the original or the updated version of the file is lost during crashes.

6.5 Discussing the Implementation

This section is intended mainly to discuss a few relevant implementation points and share our experiences while developing the mechanisms described previously and provide some insight on the relationship of NVM with today's operating systems. These are mostly effects of accessing NVM as a file system and may be explored in future works. In this work we try to keep our implementation as simple as possible, mostly to allow ourselves to isolate and explore the most relevant points of the solution, such as feasibility (in terms of both performance and complexity) and usability but also to ensure the portability of our methods. We do, however, believe that the our solution may be significantly improved from more complex and sophisticated mechanisms and we hope to pursue these benefits in the future.

The first and perhaps most important topic, is about the challenge of sharing data through multiple process spaces with COW mappings. As described previously, the COW mapping behaves similarly to private mappings in traditional file systems. In both cases, during the page fault, the page being accessed (for write) has its content copied to a new page and added into the owner process address space. However this page is not kept in the page cache and is not shared by the other processes, thus the page is not visible to other processes mapping the same file. This means that every process mapping the file using the COW mapping technique will have its own version of the pages in volatile memory, which does not happens in traditional or direct file mappings (and

consequentially not on atomic mappings). One way to allow multiple process to share data using the same COW mapping is by sharing page table entries which is not a new concept and already have been studied for other purposes [84]. Also, A persistent page table-like structure for files is described in [107] which could be an effective alternative that also reduces the overhead of mapping files into applications address space.

On the other hand, allowing multiple processes to write to the same page of the same file may risk file data consistency in some cases. For example, if two processes A and B write to the same page X and this single page is shared among them, once process A flushes its updates on X to PMFS, X is also carrying the modifications made by B. Since the data written by process B may not yet be in a consistent state, once A flushes page X to NVM, the file may be in an inconsistent state. In this case, if B fails for any reason, the file will be dirty and PMFS will not be able to recover it. There is no easy solution for these cases. As it is in other solutions, applications have to manage access to critical areas and avoid concurrency races themselves [128] otherwise consistency may be compromised.

In Section 7 we compare the performance of our solution against the existing mapping in PMFS and analyze the results. One characteristic that directly impacts these results is that, in PMFS direct mappings, whenever the *msync* system call is issued, since there are no pages in memory to be flushed in this case, all the file system does is flush any remaining data in cache. This may be expensive for the application but is necessary to ensure that all updates are written to the NVM. In COW mappings, however, this does not happen as the data cached on application operations (e.g. writes to the pages in DRAM) and the data stored in NVM are two separate things. When copying data from DRAM to NVM, our solution does not cache writes, but it can read the current state of data in DRAM even if it is retained in cache. It is also worth noting that, according to our tests, the cost of flushing cache lines also depends on the processor and overall architecture.

As described earlier in this work, our implemented mapping methods may need to manipulate the process page table entries directly in some cases. In order to avoid having to access the main memory every time a virtual address needs to be translated, current systems cache these entries in the Translation Look-aside Buffer (TLB). The issue here is that once we updated the state of a page table entry (e.g. marking them clean or marking them read-only), we must flush the TLB in order to synchronize these modifications otherwise the system may behave in unexpected ways. Flushing the TLB is an expensive operation and in some cases may represent a significant performance penalty.

Finally, another limitation of our solution is that it cannot allow accessing the same file pages through both a direct way (such as through atomic or direct mappings) and an out-of-place way (such as through COW mappings and COW-enabled writes) at the same time. This is because updating these pages using a copy-on-write technique may generate conflicts with methods that map the pages directly into the user's address space, since during the copy-on-write process, the original data page is released at the end. This constraint is also presented and discussed by the NOVA file system, regarding the possibility of allowing users to directly map NOVA's files.

7. EVALUATION

We evaluate our proposed file mapping implementations to both verify its correctness in keeping user data in consistent state and to determine whether it is a feasible alternative to existing access methods such as direct mappings and read/write functions. For the correctness part, we try to force file corruption by injecting errors in both user and kernel code, and check whether the file system recovery process is able to revert the file to a consistent state. For the feasibility part, we use file system benchmarks to measure the performance of both atomic and COW mappings and then compare the results to the ones obtained using regular file mapping techniques. To some extent we agree that using traditional benchmarks to measure [31] performance on NVM file systems may present some bias, however this is the methods used by most researchers today to evaluate their proposals, including the original PMFS evaluation [32][126][100][15].

7.1 NVM Emulation

As discussed earlier in this work, the idea behind our solution is partially based on the fact that NVM (in its current state) is significantly slower than today's DRAM technology, specially when it comes to write operations. More specifically, our COW mapping design employs kernel virtual memory to speed up write operations to the mapping region. However most NVM technologies are currently under development and access to machines with such technology is still very limited. Therefore we chose to emulate NVM using regular DRAM. Doing so is quite a simple task, since PMFS is already designed to be built over a pre-allocated region of the system's RAM.

Regarding the latency difference between DRAM and NVM technologies (such as PCRAM) we emulate NVM latency by artificially introducing additional overhead to regular memory access when writing to the file system region (which is built over DRAM). The idea is to adopt processor-level operations to account for the additional access time NVM technologies has in relation to DRAM. We do not consider the overhead on read operations as the disparity between DRAM and NVM in regard of read operations is much less expressive than in writes and it would also be much trickier. Since we are not necessarily aiming for high fidelity to the actual NVM bandwidth and performance in our results but instead we want to check how close to existing access methods our implementation can get, we only emulate latency for write operations.

Our emulation of NVM latency is based on the values for PCM presented in Table 2.1 and also on the Mnemosyne implementation. For our performance tests, writes to PCRAM are 4 times slower than writes to DRAM. The idea is to use processor operations to delay the processor for a limited time whenever a write to NVM occurs. We insert these operations in all the main points where PMFS writes to NVM, such as during writes to user data, inode updates, journaling, recovery process, among others. Besides Mnemosyne, our method for NVM emulation through artificial overhead is somewhat similar to the ones adopted by other studies [32][15].

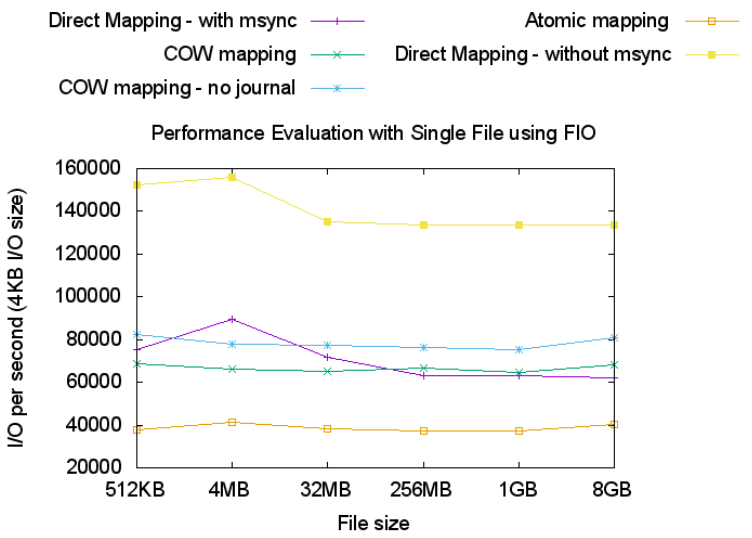

Figure 7.1: Performance of mapping and overwriting an entire file before performing *msync* and *munmap*.

7.2 Evaluating Correctness

The correctness of our implementation is determined by whether it can ensure that files manipulated by it may be restored to a consistent state at any time. To test our implementation we employ a process that continuously writes to different files and program its halt at random points of the execution. We then check the file data and its metadata for corruption and validate its content. We have performed this test over 30 times for each of 5 different files (different extensions - images, pdf and text - and size - ranging from 50-100 KB to 12-18 MB) with no file corruption in any case. This test ensures that issues that occurs at nay point during the execution of user level code will not compromise the structure of the file or the file system.

Performing the same check on kernel side is a little more complicated, as, in our case, we cannot simply halt the entire system or simulate kernel panics, as the file system data is stored in DRAM and would be completely lost during system reboot. Hence, we chose to once again forcibly halt the operating system by triggering errors in the kernel code. In this case we may need to trigger the file system recovery process either manually or by remounting PMFS as, in some cases, the file system will not replay the journal automatically. This happens, for example, when the process is locked in kernel side due to fatal errors in the kernel.

To test our kernel-side code, we chose to provoke these kernel crashes in specific places, rather then on complete random, for both practicality and simplicity reasons. This approach also helped to identify issues with the implementation and their their precise root cause, making it easy to correct them. We have selected critical points in the life-cycle of both mapping implementations to perform this evaluation, such as, to name a few: during the page fault process, during the *msync* call when the mapping is only partially flushed back to the file, after a log-entry is written but before it is committed, after a page is logged but before it is written, during a page write back among

others. We have performed well over a hundred of these tests (about 10 for each major point of the code we selected) with no files lost. With this test we ensure that failures when executing kernel code may be reverted on file system mount and will not compromise file consistency.

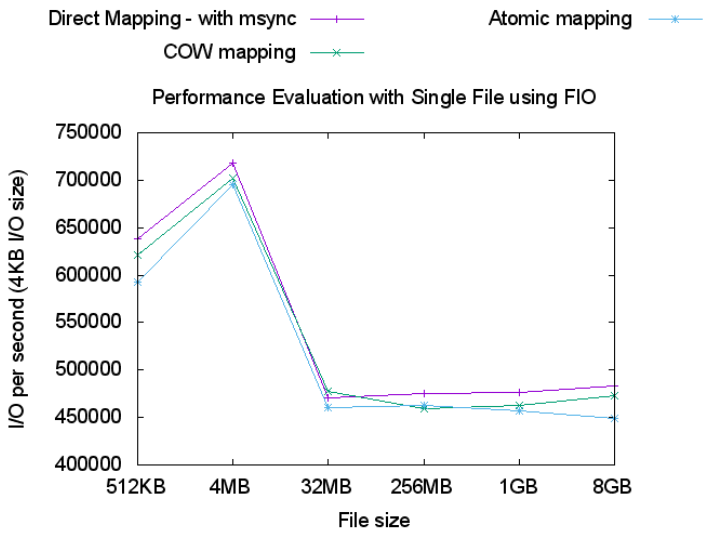

Figure 7.2: Performance of mapping and reading an entire file.

7.3 Performance evaluation

To evaluate our implementation's performance, we employ the FIO file system benchmark, mostly for its popularity and for its intuitive source code. It is also the microbenchmark used to evaluate PMFS *mmap* performance in its original paper. We run the benchmark over a HP BL620c G7 cluster node with two Intel Xeon E7 2850 2.0 GHz and 80 GB of DRAM. Our PMFs partition takes 25 GB from the DRAM space. For compatibility reasons with PMFS 3.11.0 Linux kernel, our system runs on Ubuntu 12.04.

Each test was performed from 5 to 10 times (depending on the variation of the results), during a period of 10 minutes each test. Although the machine used for these tests was allocated exclusively for benchmarking PMFS, at least once for each test case, we reboot and remount the file system to ensure no bias created by Os unexpected behavior. We believe these precautions to be enough to ensure the reliability of the results presented here.

We have also performed many (undocumented) performances tests on controlled simulated environments using virtual machines. The goal of these tests were to analyze the behavior of our solutions and to ensure our results (obtained in the real machine) are plausible. Although these results were not written down and therefore are not shown here, we believe they are worth mentioning here as these preliminary tests are much extensive in number than the ones presented here and are just as relevant to this work.

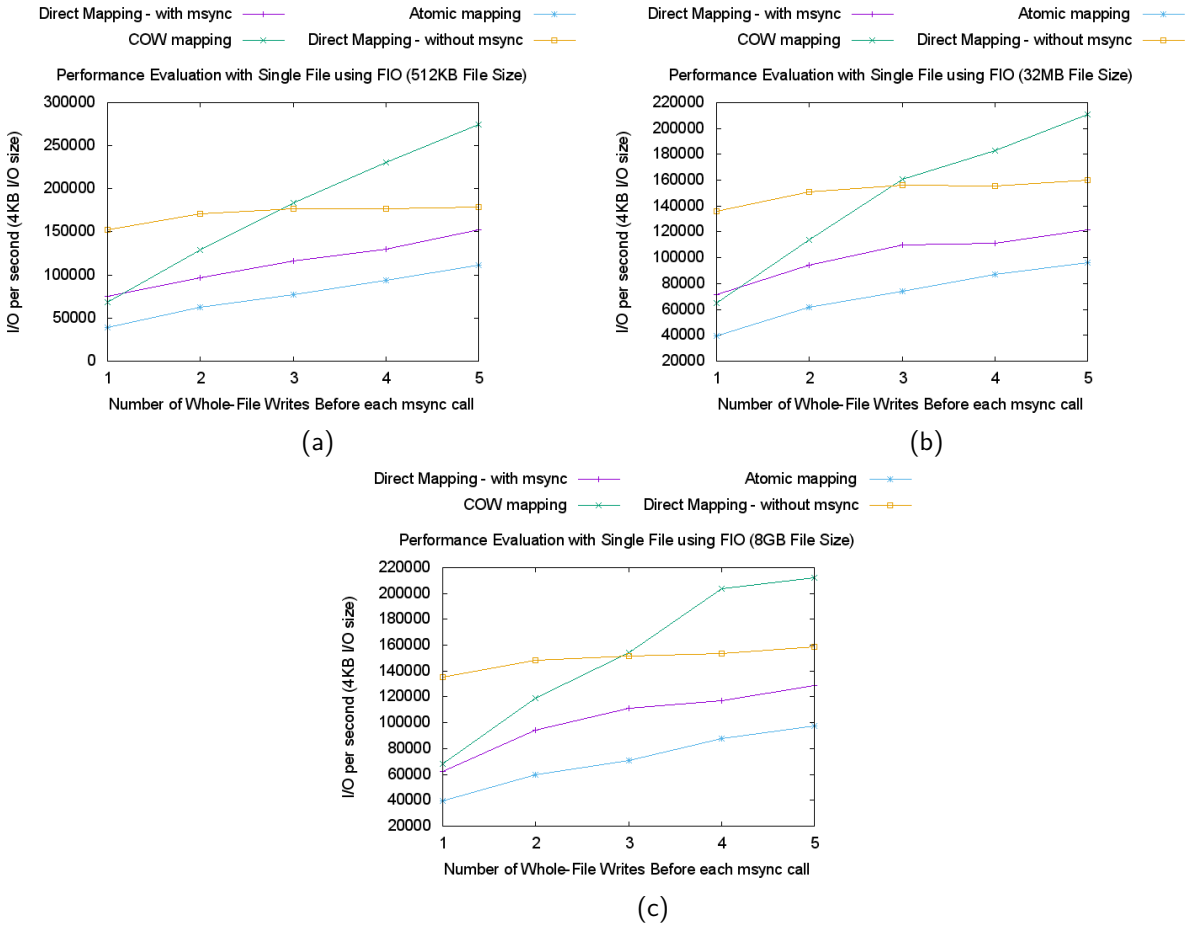

Figure 7.3: Performance results of FIO benchmark: FIO writes the whole file up to 5 times and flushes data to PMFS through *msync*.

7.3.1 Microbenchmark evaluation

We try to maintain modifications to FIO to a minimum in order to maintain its consistency and avoiding creating any kind of bias. We do however explore different aspects of our implementations by making a few modifications in FIO source code. We configure FIO to write to a file continuously and perform the *msync* and *munmap* calls in a fixed rate (usually after the file is completely written). In our initial test, each file mapping implementation writes the entire file before flushing and unmapping it. The results are shown in Figure 7.1. Unlike COW and atomic mappings, in direct mappings, *msync* call is not necessary to make writes persistent. It is, however, important to provide the guarantee that data has reached the file system in some cases, hence, in this test we evaluate direct mappings both with and without the *msync* system call at the end of each writing cycle.

The results show that COW mappings may present a performance roughly 2 times slower than direct mapping, while atomic mappings shows results almost 4 times worse in this same case. By flushing direct mappings periodically, we see that its performance drastically falls, to the point where it is more inefficient than performing COW mapping in most cases. The performance difference

floats between 27% of loss to 9% of increase at most for COW mapped files and it is evident specially on larger files where the cost of flushing the cache grows and surpasses the COW mappings cost of copy-on-write. The atomic mappings, that also performs cache flush and do not benefit from the same behavior of COW, presents between 36% to 48% performance loss when compared to direct mappings.

On the other hand, on Figure 7.2 we see the comparison on read performance only. Since all three mappings behave basically the same way on read operations, their performance is roughly the same. The small gap between them may be attributed to a small overhead during the mapping and unmapping processes, regarding transaction control. It is important to note that we do not perform the *msync* operation in this test case since it is not necessary and would not change the benchmark behavior whatsoever.

In some cases, simply allowing users to checkpoint their files by giving them a primitive to flush its mapped data to the back file may be just enough. COW mapping implements consistency in two levels at different moments: during writes to the mapped region (by writing to a page copy in DRAM instead) and during flushes to the backing file (using PMFS journal and copy-on-write techniques). We may relax the reliability constraints in Copy-On-Write mappings by skipping the PMFS side copy-on-write during the flush process, thus losing the guarantee of consistency during the execution of *msync*. Thus, in Figure 7.1 we show the impact of bypassing the PMFS copy-on-write on performance. In this figure we are able to see the overhead incurred by the file system level consistency mechanics of COW mappings, which is around 17%.

Writing to the file once and flushing it to the file system, although an acceptable and somewhat common usage of file mappings, it is not the best case scenario for both atomic and COW mappings. These solutions were designed to perform better with less frequently synchronizations with the backing file and to give the user the control over the trade-off of performance and reliability. With that in mind, we evaluate the impact of delaying the synchronization process by allowing FIO to write the files multiple times before issuing the *msync* call.

The results are presented in Figure 7.3 and show a significant improvement in the throughput and a steady growth in performance on all mapping mechanisms with larger time windows between synchronizations. In this scenario we see that the COW mapping may present results ranging from 10% and 65% of performance loss up to 110% and 58% of improvement when compared to direct mappings with and without the *msync* operation respectively. On the other hand, the performance disparity between atomic mappings and the direct mappings is situated between 43% and 18%. These numbers grow to 75% and 38% when compared to the direct mapping's results without the flush procedure.

However, how feasible it is to write these files multiple times? Naturally, writing twice an entire 8 GB file takes much more time than writing twice a 512 Kb file. Therefore, to have a more practical view on the advantages of lazily writing on mapped files with our proposed methods, we evaluate lazy writes using a fixed time windows to determine the checkpoints of the file, instead of the number of whole-file updates we have used in the previous evaluation.

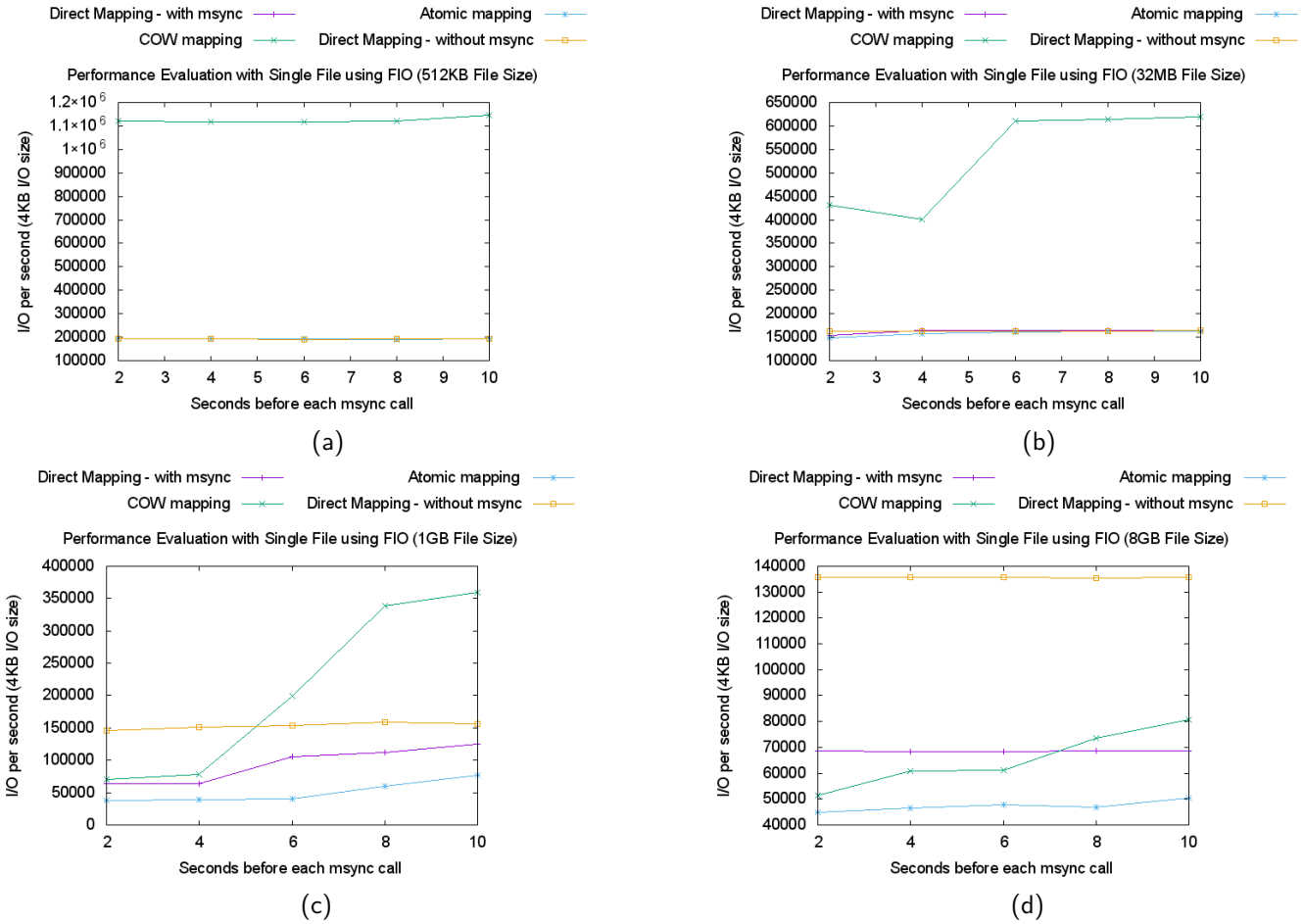

Figure 7.4: Performance results of FIO benchmark: FIO writes to a file during up to 10 seconds before committing the data to PMFS *msync*.

The results on Figure 7.4 show that, for small and medium files, we clearly have a gain in performance for larger time windows, which is specially apparent for COW mappings. In the case of atomic and direct mappings, larger time windows present significantly less impact as the throughput in both cases is mostly limited by the emulated NVM throughput and cache usage. The best example of COW mapping expected behavior shown on Figure 7.4c, with 1 GB file size, where it becomes clear that the first write to the mapped file is much slower than the subsequent writes. For larger files, the performance of all mapping models drop drastically, as the amount of pages being written twice becomes smaller and the performance gets closer to that of writing the file only once before flushing it.

Figure 7.4 gives us an idea of the applicability of lazily flushing modifications to PMFS in order to increase efficiency. Naturally, the actual amounts of time needed to obtain any relevant performance gains depends greatly on the workload behavior and throughput of the NVM technology, among other things. The graphic also illustrates the amount of data the applications may risk losing during crashes, by delaying the commits to PMFS file. This is yet another detail developers must be aware of when designing their applications.

7.3.2 Uniform memory evaluation

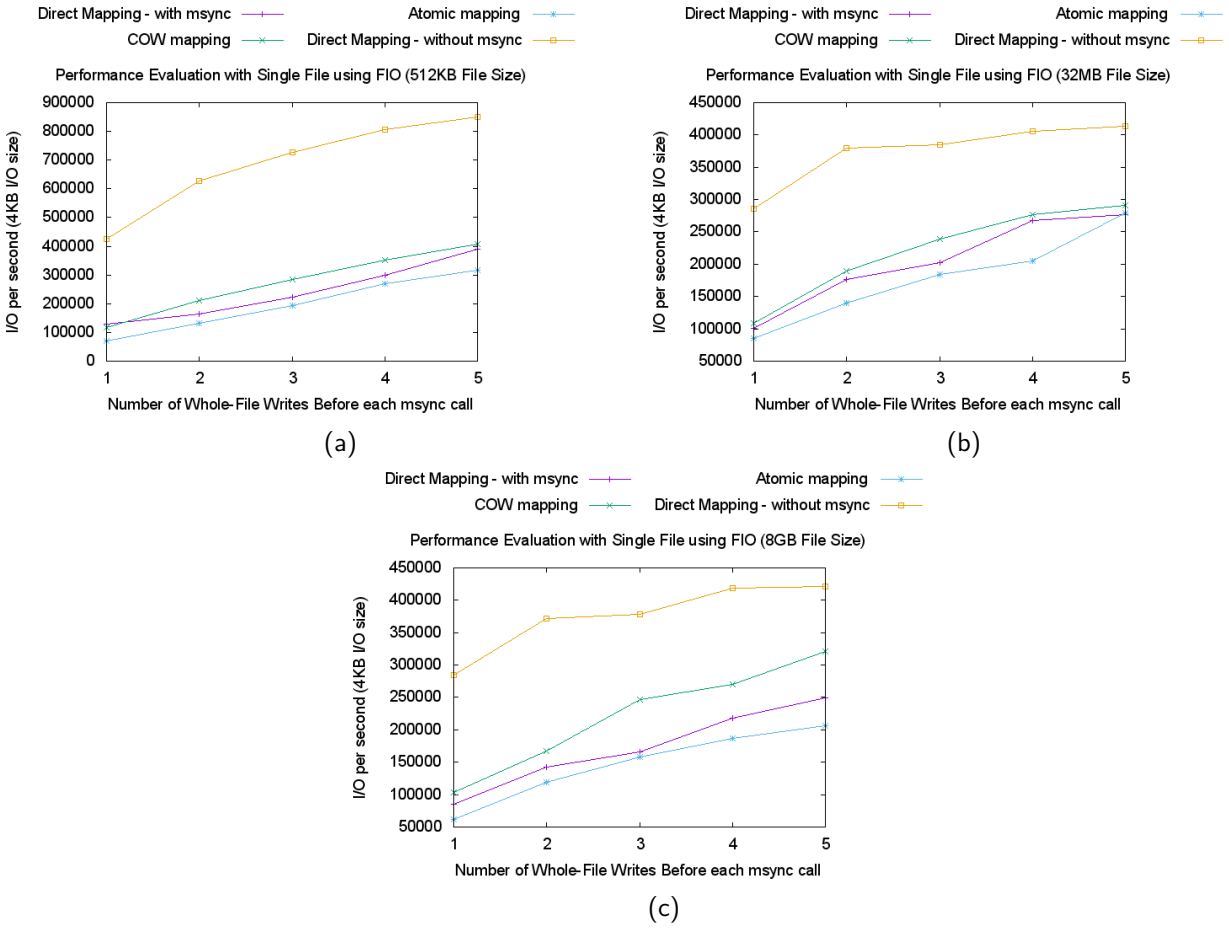

Figure 7.5: Performance results of FIO benchmark without NVM latency simulation.

In the previous section we evaluated COW and atomic mapping on a hybrid memory environment (where DRAM coexists with NVM), by simulating NVM latency. In this section we abandon the NVM simulation code and evaluate our implementation on a uniform memory technology environment. With this setup we expect to comparatively measure how our solution would behave in a pure NVM system. With this environment we also want to give atomic mappings a fair performance evaluation, as its design was greatly driven by the possibility of a NVM-only architecture. In these conditions we reapply the tests with the FIO benchmark and analyze the results accordingly.

In Figure 7.5 we see the results of our experiment. It shows that the performance of our proposed implementations are very close to that of flushed direct mappings. In terms of efficiency, this is the worst case scenario for our solutions: the file's data is replicated once for atomic mappings (during the creation of the safe copy) and twice on the COW mapping (once during while faulting the pages into the process address space and once during the *msync* call) with no faster memory to be used as buffer. This is reflected on the results, as our methods can get up to 6 times slower than direct mapping (the version with no *msync*), specially for small files.

One thing that calls our attention on Figure 7.5 is that atomic mappings still significantly slower than our COW-based solution, even though it clearly performs less work. We credit that once again to the process of cache flush. Atomic mappings has to force every cache line related to the address range given to *msync* to be flushed, in order to ensure all data is persisted before committing the transaction. As we have seen with direct mappings previously, this process may be very expensive and represent a serious performance penalty. As we have discussed earlier in this work, improving efficiency of cache with NVM-based storage is one of the main topics of research in NVM and some solutions have already been developed. It is also worth noting that the cost of this operation depends greatly on the architecture.

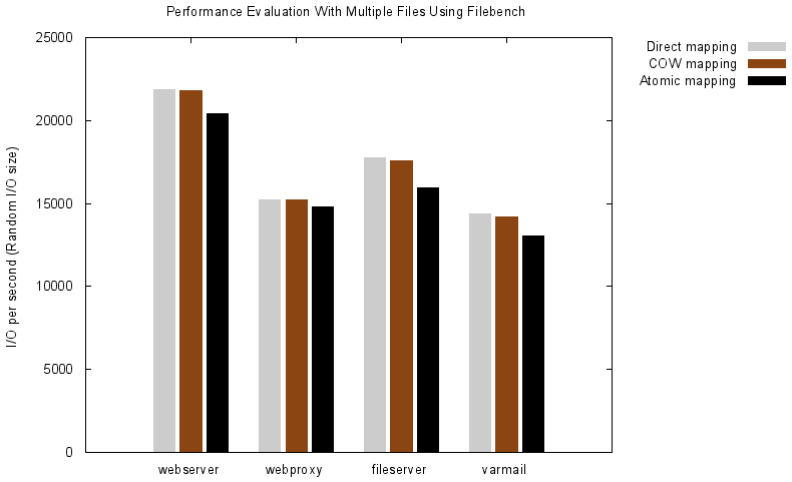

Figure 7.6: Performance of mapping and reading and entire file.

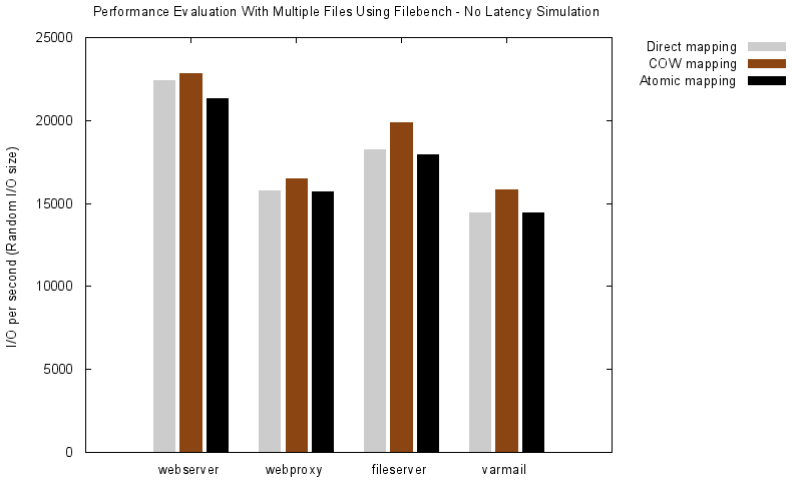

Figure 7.7: Performance of mapping and reading and entire file.

7.3.3 Macrobenchmark evaluation

To complement the results obtained previously with the FIO microbenchmark we also perform macrobenchmark tests, in order to evaluate the efficiency of our solution when used with common real-world workloads. For our macrobenchmark evaluation, we use the well-known filebench benchmark. We chose filebench due to its flexibility and simple structure. We modify filebench to use memory mapped files for its operations and design a few workloads based on traditional workloads shipped with filebench. Unlike FIO in our previous evaluation, these workloads perform combinations of reads, writes and appends and manipulate thousands of small files.

We show the results of filebench execution in Figure 7.6. Similar to our microbenchmarks results, these graphics show that the performance penalty of our proposed solution is very close to that of calling *msync* in order to ensure data reaches the storage. The throughput difference between direct mapping and atomic mapping (which is our slower implementation, as we have seen in previous tests) is roughly 11.5% in the worst case.

We also apply filebench evaluation on uniform memory model in order to validate our implementation on a a single memory technology environment. The results are shown in Figure 7.7. Even without the additional overhead of writing to a slower memory, we see little difference on the results. This is the reflection of combining small writes of low overhead with small reads that have near to zero overhead. Although the solutions proposed in this paper were aimed at a different scenario (namely systems that map and manage large chunks of NVM), these results are still useful to illustrate the the impacts as well as the versatility of such simple mechanisms.

8. CONCLUSION

In this study, we presented two methods of accessing data stored in NVM file systems based on the already well established XIP-enabled memory mapped file concept. Both are methods that enforce file persistence and consistency by delegating to the user the decision of when data must be persisted and giving him the ability to perform checkpoints over its own files. Our goal with this work is to provide to users and framework developers the means to securely write to NVM file systems without having to implement their own mechanisms. We also study the advantages of using DRAM as a buffer for file mapped pages in order to compensate for our solution's additional overhead.

We have demonstrated that our solutions can guarantee file consistency even during unexpected system halts by adopting simple techniques such as out-of-place updates and copy-on-write. The performance penalty for incorporating these reliability mechanisms on memory mapped files may vary greatly, depending on how users employ these mapping methods. We also successfully mitigated this performance penalty by adopting DRAM as write buffer for memory mapped files, to the point of obtaining significant performance improvement over directly mapped files with no reliability guarantees in some cases. Through the implementation of COW and atomic mappings, we also managed to observe a couple behavioral characteristics of file access on NVM, such as the impact of cache flushing on large sequential writes. Additionally, our implementation, although straightforward and effective, presents some limitations, specially regarding file sharing across multiple processes. Although we agree that these topics are worth exploring further, we leave them for future work.

We have also thoroughly surveyed current research in the NVM file system area, attempting to identify the relevant topics and common challenges and solution designs. Although these studies present various interesting proposals and insights, there are no definitive approaches, even though some trends can already be identified. For instance, one matter that is currently receiving a significant amount of attention is the one regarding NVM scalability. Because of its superior density (compared to DRAM), NVM is expected to be employed in large amounts in future systems. However, today's operating systems (and processors) are not yet prepared to manage such large memories. This memory pool is expected to grow to the level of petabytes, which, consequentially, leads to many concerns regarding the scalability of these architectures, especially when treating details like memory addressing, cache coherency and handling failure to memory accesses.

Migrating current concepts and mechanics to an NVM-enabled architecture is going to be an iterative process and much work has to be done until the memory hierarchy is adjusted to this new paradigm. Although it may seem that NVM storage is a distant reality sometimes, existing projects are already available and aim to transparently provide current systems access to persistent memories. These are the first steps towards a more efficient and reliable memory architecture that will explore interfaces beyond the file system abstraction unleashing the full disruptive potential of NVM systems.

Finally, it should be clear that we believe that mapping file system regions will be an important mechanism for both applications and tools that aim to work with NVM efficiently. Thus, optimizing this type of access is very important and it is currently a common target of research in the area of NVM [121][108][128][32]. We expect this optimization to touch various aspects of memory management and file system like more efficient addressing solutions, alternative file system consistency methods and more fine grained data persistence.

BIBLIOGRAPHY

[1] Ames, A., Maltzahn, C., Bobb, N., Miller, E. L., Brandt, S. A., Neeman, A., Hiatt, A., and Tuteja, D. Richer file system metadata using links and attributes. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems Technologies*, pages 49–60, 2005.

[2] Baek, S., Choi, J., Ahn, S., Lee, D., and Noh, S. H. Design and implementation of a uniformity-improving page allocation scheme for flash-based storage systems. *Design Automation for Embedded Systems*, 13(2):5–25, Jun 2009.

[3] Baek, S., Son, D., Kang, D., Choi, J., and Cho, S. Design Space Exploration of an NVM-based Memory Hierarchy. In *Proceedings of the 32nd IEEE International Conference on Computer Design*, pages 224–229, 2014.

[4] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot topics in operating systems*, pages 2–2, 2011.

[5] Barbosa, O. and Alves, C. A Systematic Mapping Study on Software Ecosystems. In *Proceedings of the 2011 Workshop on Software Ecosystems*, pages 15–26, 2011.

[6] Bhandari, K., Chakrabarti, D. R., and Boehm, H.-J. Implications of cpu caching on byte-addressable non-volatile memory programming. *Technical Report, Hewlett-Packard, HPL-2012-236*, page 7, 2012.

[7] Biswas, P. and Towsley, D. Performance Analysis of Distributed File Systems with Non-volatile Caches. In *Proceedings of the 2nd IEEE International Symposium on High Performance Distributed Computing*, pages 252–262, 1993.

[8] Burr, G. W., Kurdi, B. N., Scott, J. C., Lam, C. H., Gopalakrishnan, K., and Shenoy, R. S. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52(4):449–464, Jul 2008.

[9] Caulfield, A. M., Coburn, J., Mollov, T. I., De, A., Akel, A., He, J., Jagatheesan, A., Gupta, R. K., Snavely, A., and Swanson, S. Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[10] Caulfield, A. M., Mollov, T. I., Eisner, L. A., De, A., Coburn, J., and Swanson, S. Providing safe, user space access to fast, solid state disks. *ACM Special Interest Group on Computer Architecture News*, 40(1):387–400, Mar 2012.

[11] Caulfield, A. M. and Swanson, S. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th ACM Annual International Symposium on Computer Architecture*, pages 464–474, 2013.

[12] Chang, H.-s., Chang, Y.-h., Hsiu, P.-c., Kuo, T.-w., and Li, H.-p. Marching-Based Wear-Leveling for PCM-Based Storage Systems. *ACM Transactions on Design Automation of Electronic Systems*, 20(2):25, Feb 2015.

[13] Chang, L.-p., Sung, P.-h., and Chen, P.-h. Fast File Synching for Applications in Flash-Based Android Devices. In *Proceedings of the 2014 IEEE Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2014.

[14] Chen, F., Koufaty, D. A., and Zhang, X. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM Special Interest Group on Computer Systems Performance Evaluation Review*, 37(1):181–192, Jun 2009.

[15] Chen, F., Mesnier, M. P., and Hahn, S. A Protected Block Device for Persistent Memory. In *Proceedings of the 30th IEEE Symposium on Mass Storage Systems Technologies*, pages 1–12, 2014.

[16] Chen, J., Wei, Q., Chen, C., and Wu, L. FSMAC: A file system metadata accelerator with non-volatile memory. In *Proceedings of the 2013 IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–11, 2013.

[17] Chen, K., Bunt, R. B., and Eager, D. L. Write caching in distributed file systems. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 457–466, 1995.

[18] Chen, R., Wang, Y., Hu, J., Liu, D., Shao, Z., and Guan, Y. Virtual-Machine Metadata Optimization for I/O Traffic Reduction in Mobile Virtualization. In *Proceedings of the 2014 IEEE Non-Volatile Memory Systems and Applications Symposium*, pages 1–2, 2014.

[19] Chen, Z., Lu, Y., Xiao, N., and Liu, F. A hybrid memory built by SSD and DRAM to support in-memory Big Data analytics. *Knowledge and Information Systems*, 41(2):335–354, Jan 2014.

[20] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. *ACM Special Interest Group on Programming Languages Notices*, 46(3):105–118, Mar 2011.

[21] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Special Interest Group on Operating System symposium on Operating systems principles*, pages 133–146, 2009.

[22] Corbet, J. . LFCS: Preparing linux for nonvolatile memory devices. Captured in: http://lwn.net/Articles/547903/, May 2015.

[23] Corbet, J. Supporting filesystems in persistent memory. Captured in: http://lwn.net/Articles/610174/, May 2015.

[24] Corbet., J. Memory protection keys. Captured in: http://lwn.net/Articles/643797/, October 2015.

[25] Czurylo , K. and Rudoff, A. NVM Library. Captured in: http://github.com/pmem/nvml/, March 2016.

[26] Dai, H., Neufeld, M., and Han, R. ELF: an efficient log-structured flash file system for micro sensor nodes. In *Proceedings of the 2nd ACM international Conference on Embedded networked sensor systems*, pages 176–187, 2004.

[27] Dai, P., Zhuge, Q., Chen, X., Jiang, W., and Sha, E. H.-M. Effective file data-block placement for different types of page cache on hybrid main memory architectures. *Design Automation for Embedded Systems*, 17(3-4):485–506, Sep 2014.

[28] Doh, I. H., Choi, J., Lee, D., and Noh, S. H. Exploiting Non-Volatile RAM to Enhance Flash File System Performance. In *Proceedings of the 7th ACM & IEEE international Conference on Embedded software*, pages 164–173, 2007.

[29] Doh, I. H., Choi, J., Lee, D., and Noh, S. H. An Empirical Study of Deploying Storage Class Memory into the I/O Path of Portable Systems. *The Computer Journal*, 54(8):1267–1281, Aug 2011.

[30] Doh, I. H., Lee, H. J., Moon, Y. J., Kim, E., Choi, J., Lee, D., and Noh, S. H. Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, pages 1658–1663, 2009.

[31] Dong, M., Yu, Q., Zhou, X., Hong, Y., Chen, H., and Zang, B. Rethinking benchmarking for nvm-based file systems. In *Proceedings of the 7th ACM Special Interest Group on Operating Systems Asia-Pacific Workshop on Systems*, page 20, 2016.

[32] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J. System software for persistent memory. In *Proceedings of the 9th ACM European Conference on Computer Systems*, pages 1–15, 2014.

[33] Faraboschi, P., Keeton, K., Marsland, T., and Milojicic, D. Beyond processor-centric operating systems. In *Proceedings of the 15th USENIX Workshop on Hot Topics in Operating Systems*, pages 1–7, 2015.

[34] Freitas, R. F. and Wilcke, W. W. Storage-class memory: The next storage system technology. *IBM Journal of Research and Development*, 52(4.5):439–447, Jul 2008.

[35] Fu, M.-M. and Dasgupta, P. A concurrent programming environment for memory-mapped persistent object systems. In *Proceedings of the 17th IEEE International Computer Software and Applications Conference*, pages 291–298, 1993.

[36] Gal, E. and Toledo, S. Mapping structures for flash memories: Techniques and open problems. In *Proceedings of the 2005 IEEE International Conference on Software-Science, Technology and Engineering*, pages 83–92, 2005.

[37] Giles, E., Doshi, K., and Varman, P. Bridging the programming gap between persistent and volatile memory using WrAP. In *Proceedings of the 2013 ACM International Conference on Computing Frontiers*, pages 1–10, 2013.

[38] Greenan, K. M. and Miller, E. L. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proceedings of the 6th ACM & IEEE International Conference on Embedded software*, pages 178–187, 2006.

[39] Han, J.-il, Kim, Y.-m., and Lee, J. Achieving Energy-Efficiency with a Next Generation NVRAM-based SSD. In *Proceedings of the 2014 IEEE International Conference on Information and Communication Technology Convergence*, pages 563–568, 2014.

[40] Hongwei, Z., Xuehua, Z., and Bao'an, Z. Storage Management Strategy of Flash-Based Platform. *Transactions of Tianjin University*, 15(1):70–74, Sep 2009.

[41] Hu, Y., Nightingale, T., and Yang, Q. RAPID-CacheĐA Reliable and Inexpensive Write Cache for High Performance Storage Systems. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):290–307, Mar 2002.

[42] Huang, J., Schwan, K., and Qureshi, M. NVRAM-aware Logging in Transaction Systems. In *Proceedings of the 2014 VLDB Endowment*, pages 389–400, 2014.

[43] Hwang, T., Jung, J., and Won, Y. HEAPO: Heap-Based Persistent Object Store. In *Proceedings of the 2014 VLDB Endowment*, pages 1–21, 2014.

[44] Hwang, Y., Gwak, H., and Shin, D. Two-Level Logging with Non-Volatile Byte-Addressable Memory in Log-Structured File Systems. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, page 38, 2015.

[45] Im, S. and Shin, D. Differentiated space allocation for wear leveling on phase-change memory-based storage device. *IEEE Transactions on Consumer Electronics*, 60(1):45–51, Apr 2014.

[46] Islam, N. S., Wasi-ur-Rahman, Md., Lu, X., and Panda, D. K. High Performance Design for HDFS with Byte-Addressability of NVM and RDMA. In *Proceedings of the 2016 ACM International Conference on Supercomputing*, page 8, 2016.

[47] Joint Electron Tube Engineering Council JEDEC. JEDEC Byte-Addressable Energy Backed Interface. Captured in: http://www.jedec.org/standards-documents/docs/jesd245, March 2016.

[48] Jeremic, N., Mühl, G., Busse, A., and Richling, J. Operating system support for dynamic over-provisioning of solid state drives. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1753–1758, 2012.

[49] Jin, R., Cho, H.-J., Lee, S.-W., and Chung, T.-S. Lazy-split B+-tree: a novel B+-tree index scheme for flash-based database systems. *Design Automation for Embedded Systems*, 17(1):167–191, Nov 2013.

[50] Johnson, T. and Davis, T. A. Parallel buddy memory management. *Parallel Processing Letters*, 2(4):391–398, Dec 1992.

[51] Jung, J., Won, Y., Kim, E., Shin, H., and Jeon, B. Frash: Exploiting storage class memory in hybrid file system for hierarchical storage. *ACM Transactions on Storage*, 6(1):3, Mar 2010.

[52] Jung, M., Wilson, E. H., Choi, W., Shalf, J., Aktulga, H. M., Yang, C., Saule, E., Catalyurek, U. V., and Kandemir, M. Exploring the future of out-of-core computing with compute-local non-volatile memory. In *Proceedings of the 12th ACM International Conference on Computing Frontiers*, pages 125–139, 2014.

[53] Kang, D., Baek, S., Choi, J., Lee, D., Noh, S. H., and Mutlu, O. Amnesic Cache Management for Non-Volatile Memory. In *Proceedings of the 31st IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–13, 2015.

[54] Kang, S., Park, S., Jung, H., Shim, H., and Cha, J. Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices. *IEEE Transactions on Computers*, 58(6):744–758, Jun 2009.

[55] Kang, Y., Pitchumani, R., Marlette, T., and Miller, E. L. Muninn: a Versioning Flash Key-Value Store Using an Object-based Storage Model. In *Proceedings of the 2014 ACM International Conference on Systems and Storage*, pages 1–11, 2014.

[56] Kannan, S., Gavrilovska, A., and Schwan, K. pvm: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the 11th ACM European Conference on Computer Systems*, page 13, 2016.

[57] Kannan, S., Gavrilovska, A., Schwan, K., Milojicic, D., and Talwar, V. Using active NVRAM for I/O staging. In *Proceedings of the 2nd ACM international Workshop on Petascal data analytics: challenges and opportunities*, pages 15–22, 2011.

[58] Keele, S. Guidelines for performing Systematic Literature Reviews in Software Engineering. *Technical Report, EBSE*, page 7, Jul 2007.

[59] Khatib, M. G., Hartel, P. H., and Van Dijk, H. W. Energy-efficient streaming using non-volatile memory. *Journal of Signal Processing Systems*, 60(2):149–168, Aug 2010.

[60] Kim, D. and Kang, S. Dual Region Write Buffering: Making Large-Scale Nonvolatile Buffer using Small Capacitor in SSD. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2039–2046, 2015.

[61] Kim, H. In-Memory File System for Non-Volatile Memory. In *Proceedings of the 2013 ACM Research in Adaptive and Convergent Systems*, pages 479–484, 2013.

[62] Kim, J., Min, C., and Eom, Y. I. Reducing Excessive Journaling Overhead with Small-Sized NVRAM for Mobile Devices. *IEEE Transactions on Consumer Electronics*, 60(2):217–224, May 2014.

[63] Kim, Y., Gupta, A., and Urgaonkar, B. A Temporal Locality-Aware Page-Mapped Flash Translation Layer. *Journal of Computer Science and Technology*, 28(6):1025–1044, Nov 2013.

[64] Komalan, M. P., Pérez, J. I. G., Tenllado, C., Montañana, J. M., Artés, A., Fernández, J. F. T., and Catthoor, F. Design exploration of a NVM based hybrid instruction memory organization for embedded platforms. *Design Automation for Embedded Systems*, 17(3-4):459–483, Oct 2014.

[65] Kryder, M. H. and Kim, C. S. After Hard Drives - What Comes Next? *IEEE Transactions on Magnetics*, 45(10):3406–3413, Oct 2009.

[66] Kwon, S. J., Ranjitkar, A., Ko, Y.-B., and Chung, Tae-Sun. FTL algorithms for NAND-type flash memories. *Design Automation for Embedded Systems*, 15(3):191–224, Mar 2011.

[67] Lee, E., Bahn, H., and Noh, S. H. A Unified Buffer Cache Architecture that Subsumes Journaling Functionality via Nonvolatile Memory. *ACM Transactions on Storage*, 10(1):1–17, Jan 2014.

[68] Lee, E., Kang, H., Bahn, H., and Shin, K. Eliminating Periodic Flush Overhead of File I/O with Non-volatile Buffer Cache. *IEEE Transactions on Computers*, 65(4):1145–1157, Apr 2016.

[69] Lee, E., Yoo, S., Jang, J., and Bahn, H. WIPS: a write-in-place snapshot file system for storage-class memory. *Electronics letters*, 48(17):16–17, Aug 2012.

[70] Lee, H. High-performance NAND and PRAM hybrid storage design for consumer electronics. *IEEE Transactions on Consumer Electronics*, 56(1):112–118, Feb 2010.

[71] Lee, K., Ryu, S., and Han, H. Performance implications of cache flushes for non-volatile memory file systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 2069–2071, 2015.

[72] Lee, S., Kim, J., Lee, M., Lee, H., and Eom, Y. . Last Block Logging Mechanism for Improving Performance and Lifetime on SCM-based File System. In *Proceedings of the 8th ACM International Conference on Ubiquitous Information Management and Communication*, page 38, 2014.

[73] Lee, Y., Jung, S., Choi, M., and Song, Y. H. An efficient management scheme for updating redundant information in flash-based storage system. *Design Automation for Embedded Systems*, 14(4):389–413, Nov 2010.

[74] Li, D., Liao, X., Jin, H., Tang, Y., and Zhao, G. Writeback throttling in a virtualized system with SCM. *Frontiers of Computer Science*, 10(1):82–95, Jun 2016.

[75] Li, G., Zhao, P., Yuan, L., and Gao, S. Efficient implementation of a multi-dimensional index structure over flash memory storage systems. *The Journal of Supercomputing*, 64(3):1055–1074, Sep 2011.

[76] Li, H.-y., Xiong, N.-x., Huang, P., and Gui, C. PASS: a simple, efficient parallelism-aware solid state drive I/O scheduler. *Journal of Zhejiang University SCIENCE C*, 15(5):321–336, May 2014.

[77] Li, J., Zhuge, Q., Liu, D., Luo, H., and Sha, E. H.-M. A content-aware writing mechanism for reducing energy on non-volatile memory based embedded storage systems. *Design Automation for Embedded Systems*, 17(3-4):711–737, Oct 2014.

[78] Li, X. and Lu, K. FCKPT:A fine-grained incremental checkpoint for PCM. In *Proceedings of the 2011 IEEE International Conference on Computer Science and Network Technology*, pages 2019–2023, 2011.

[79] Li, Y., Wang, Y., Jiang, A., and Bruck, J. Content-assisted file decoding for nonvolatile memories. In *Proceedings of the 2012 IEEE Asilomar Conference on Signals, Systems and Computers*, pages 937–941, 2012.

[80] Lim, S.-H. and Seo, M.-K. Efficient non-linear multimedia editing for non-volatile mobile storage. In *Proceedings of the 2010 ACM Workshop on Mobile Cloud Media Computing*, pages 59–64, 2010.

[81] Lo, S.-w. Swap-before-Hibernate : A Time Efficient Method to Suspend an OS to a Flash drive. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 201–205, 2010.

[82] Lu, Y., Eleftheriou, E., Haas, R., Iliadis, I., and Pletka, R. Extending the Lifetime of Flash-based Storage through Reducing Write Amplification from File Systems. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, pages 257–270, 2009.

[83] Lu, Y., Shu, J., and Zhu, P. TxCache: Transactional Cache using Byte-Addressable Non-Volatile Memories in SSDs. In *Proceedings of the 2014 IEEE Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2014.

[84] McCracken, D. Shared page tables redux. In *Proceedings of the 2016 Linux Symposium*, page 117, 2006.

[85] Meena, J., Sze, S., Chand, U., and Tseng, T.-Y. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9(1):526, Dec 2014.

[86] Micheloni, R., Picca, M., Amato, S., Schwalm, H., Scheppler, M., and Commodaro, S. Non-volatile memories for removable media. In *Proceedings of the IEEE*, pages 148–160, 2009.

[87] Mittal, S. and Vetter, J. A survey of software techniques for using non-volatile memories for storage and main memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1537–1550, May 2016.

[88] Nath, S. and Gibbons, P. B. Online maintenance of very large random samples on flash storage. In *Proceedings of the 2010 VLDB Endowment*, number 1, pages 67–90, 2010.

[89] Oikawa, S. Integration Methods of Main Memory and File System Management for Non-Volatile Main Memory and Implications of File System Structures. In *Proceedings of the 16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 1–8, 2013.

[90] Oikawa, S. Non-Volatile Memory Management Methods for Non-Volatile Main Memory based on File Systems. *SpringerPlus*, 54(1):1–8, Jul 2013.

[91] Oikawa, S. Independent kernel/process checkpointing on non-volatile main memory for quick kernel rejuvenation. In *Proceedings of the 2014 International Conference on Architecture of Computing Systems*, pages 233–244, 2014.

[92] Oikawa, S. and Miki, S. File-Based memory management for non-volatile main memory. In *Proceedings of the 37th Annual IEEE Computer Software and Applications Conference*, pages 559–568, 2013.

[93] Olivier, P., Boukhobza, J., and Senn, E. Micro-benchmarking flash memory file-system wear leveling and garbage collection: A focus on initial state impact. In *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering*, pages 437–444, 2012.

[94] Olivier, P., Boukhobza, J., and Senn, E. On benchmarking embedded Linux flash file systems. *ACM Special Interest Group on Embedded Systems Review*, 9(2):43–47, Jun 2012.

[95] Park, S., Kelly, T., and Shen, K. Failure-Atomic msync(): A SimpleEfficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 225–238, 2013.

[96] Park, Y. and Park, K. H. High-performance scalable flash file system using virtual metadata storage with Phase-change RAM. *IEEE Transactions on Computers*, 60(3):321–334, Mar 2011.

[97] Perez, T., Calazans, N. L. V., and De Rose, C. A. F. System-level impacts of persistent main memory using a search engine. *Microelectronics Journal*, 45(2):211–216, Feb 2014.

[98] Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M. Systematic Mapping Studies in Software Engineering. In *Proceedings of the 12th IEEE international Conference on Evaluation and Assessment in Software Engineering*, pages 68–77, 2007.

[99] Proctor., A. . Storage in the DIMM Socket. Captured in: http://www.snia.org/forums/sssi/NVDIMM, December 2015.

[100] Qiu, S. and Reddy, a. L. N. Exploiting superpages in a nonvolatile memory file system. In *Proceedings of the 28th IEEE Symposium on Mass Storage Systems and Technologies*, pages 1–5, 2012.

[101] Qureshi, M. K., Srinivasan, V., and Rivers, J. A. Scalable high performance main memory system using phase-change memory technology. *ACM Special Interest Group on Computer Architecture News*, 37(3):24–33, Jun 2009.

[102] Ramasamy, A. S. and Karantharaj, P. File System and Storage Array Design Challenges for Flash Memory. In *Proceedings of the 2014 IEEE International Conference on Green Computing Communication and Electrical Engineering*, pages 1–8, 2014.

[103] Ramos, L. and Bianchini, R. Exploiting phase-change memory in cooperative caches. In *Proceedings of the 24th IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 227–234, 2012.

[104] Raoux, S., Burr, G. W., Breitwisch, M. J., Rettner, C. T., Chen, Y.-C., Shelby, R. M., Salinga, M., Krebs, D., Chen, S.-H., and Lung, H.-L. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(45):465–479, Jul 2008.

[105] Ryu, S., Lee, K., and Han, H. In-memory Write-ahead Logging for Mobile Smart Devices with NVRAM. *IEEE Transactions on Consumer Electronics*, 61(1):39–46, Feb 2015.

[106] Seo, D. and Shin, D. WAM: Wear-Out-Aware Memory Management for SCRAM-Based Low Power Mobile Systems. *IEEE Transactions on Consumer Electronics*, 59(4):803–810, Nov 2013.

[107] Sha, E., Chen, X., Zhuge, Q., Shi, L., and Jiang, W. A new design of in-memory file system based on file virtual address framework. *IEEE Transactions on Computers*, 65(10):1–1, Oct 2016.

[108] Sha, E. H.-M., Jia, Y., Chen, X., Zhuge, Q., Jiang, W., and Qin, J. The design and implementation of an efficient user-space in-memory file system. In *Proceedings of the 5th IEEE Non-Volatile Memory Systems and Applications Symposium*, pages 1–6, 2016.

[109] Shim, H., Jung, D., Kim, J., Kim, J. S., and Maeng, S. Co-optimization of buffer layer and FTL in high-performance flash-based storage systems. *Design Automation for Embedded Systems*, 14(4):415–443, Nov 2010.

[110] Son, Y., Kang, H., Han, H., and Yeom, H. Y. An empirical evaluation and analysis of the performance of nvm express solid state drive. *Cluster Computing*, 19(3):1–13, Jul 2016.

[111] Son, Y., Song, N. Y., Han, H., Eom, H., and Yeom, H. Y. Design and evaluation of a user-level file system for fast storage devices. *Cluster Computing*, 18(3):1075–1086, Jul 2015.

[112] Sun, G., Joo, Y., Chen, Y., Niu, D., Xie, Y., Chen, Y., and Li, H. A Hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*, pages 1–12, 2010.

[113] Sun, J., Long, X., Wan, H., and Yang, J. A Checkpointing and Instant-on Mechanism for a Embedded System Based on Non-Volatile Memories. In *Proceedings of the 2014 IEEE Computing, Communications and IT Applications Conference*, pages 173 – 178, 2014.

[114] Tuteja, D., Miller, E. L., Brandt, S. A., and Edel, N. K. MRAMFS : A Compressing File System for Non-volatile RAM. In *Proceedings of the 12th IEEE Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 596–603, 2011.

[115] Unified Extensible Firmware Interface UEFI. Advanced Configuration and Power Interface Specification. Captured in: http://uefi.org/specifications, March 2016.

[116] Verma, R., Mendez, A. A., Park, S., Mannarswamy, S. S., Kelly, T. P., and Morrey III, C. B. . Failure-atomic updates of application data in a linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 203–211, 2015.

[117] Verneer, D. eXecute-in-place. *Memory Card Magazine*, Mar 1991.

[118] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne : Lightweight Persistent Memory. *Architecture Support for Programming Language and Operating System*, 39(1):91–104, Mar 2011.

[119] Wang, A.-I., Reiher, P. L., Popek, G. J., and Kuenning, G. H. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. In *Proceedings of the 2002 USENIX Annual Technical Conference, General Track*, pages 15–28, 2002.

[120] Wang, C. and Baskiyar, S. Extending flash lifetime in secondary storage. *Microprocessors and Microsystems*, 39(3):167–180, May 2015.

[121] Wilcox, M. DAX: Page cache bypass for filesystems on memory storage. Captured in: https://lwn.net/Articles/618064/, May 2015.

[122] Williams, D. ND: NFIT-Defined / NVDIMM Subsystem. Captured in: http://lwn.net/Articles/640891/, March 2016.

[123] Wu, C., Zhang, G., and Li, K. Rethinking computer architectures and software systems for phase-change memory. *ACM Journal on Emerging Technologies in Computing Systems*, 12(4):33, Jul 2016.

[124] Wu, C. H., Chang, W. Y., and Hong, Z. W. A reliable non-volatile memory system: Exploiting file-system characteristics. In *Proceedings of the 15th IEEE Pacific Rim International Symposium on Dependable Computing*, pages 202–207, 2009.

[125] Wu, C. H., Wu, P. H., Chen, K. L., Chang, W. Y., and Lai, K. C. A Hotness Filter of Files for Reliable Non-Volatile Memory Systems. *IEEE Transactions on Dependable and Secure Computing*, 12(4):375–386, Jul-Aug 2015.

[126] Wu, X. and Reddy, a. L. N. SCMFS: A file system for Storage Class Memory. In *Proceedings of the 2011 ACM International Conference on High Performance Computing, Networking, Storage Analysis*, pages 1–11, 2011.

[127] Xia, F., Jiang, D.-J., Xiong, J., and Sun, N.-H. A Survey of Phase Change Memory Systems. *Journal of Computer Science and Technology*, 30(1):121–144, Jan 2015.

[128] Xu,J. and Swanson, S. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, pages 323–338, 2016.

[129] Y. Park, S.-H., Lim, C. L., and Park, K. PFFS: a scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash. In *Proceedings of the 2008 ACM Symposium on Applied computing*, pages 1498–1503, 2008.

[130] Yang, J. J., Pickett, M. D., Li, X., Ohlberg, D. A. A., Stewart, D. R., and Williams, R. S. Memristive switching mechanism for metal/oxide/metal nanodevices. *Nature nanotechnology*, 3(7):429–433, Jul 2008.

[131] Yoon, S.-K., Bian, M.-Y., and Kim, S.-D. An integrated memory-disk system with buffering adapter and non-volatile memory. *Design Automation for Embedded Systems*, 17(4):609–626, Oct 2014.

[132] Zhao, J., Li, S., Yoon, Dh., Xie, Y., and Jouppi, Np. Kiln: closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2013.

[133] Zwisler, R. Add persistent memory driver. Captured in: http://lwn.net/Articles/609755/, March 2016.