IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received December 18, 2018, accepted January 8, 2019, date of publication February 14, 2019, date of current version March 8, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2899463

Non-Volatile Memory File Systems: A Survey

GIANLUCCA O. PUGLIA[1], AVELINO FRANCISCO ZORZO[1], (Member, IEEE), CÉSAR A. F. DE ROSE[1], (Member, IEEE), TACIANO D. PEREZ[1], AND DEJAN MILOJICIC[2], (Fellow, IEEE)

[1]Pontifical Catholic University of Rio Grande do Sul, Faculty of Informatics -FACIN, Porto Alegre 90619-900, Brazil
[2]Hewlett-Packard Labs, Palo Alto, CA 94304-1126, USA

Corresponding author: Avelino Francisco Zorzo (avelino.zorzo@pucrs.br)

This work was supported in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior–Brazil (CAPES) under Finance Code 001, and in part by the Hewlett Packard Enterprise (HPE) with resources under Law 8.248/91.

ABSTRACT For decades, computer architectures have treated memory and storage as separate entities. Nowadays, we watch the emergence of new memory technologies that promise to significantly change the landscape of memory systems. Byte-addressable non-volatile memory (NVM) technologies are expected to offer access latency close to that of dynamic random access memory and capacity suited for storage, resulting in storage-class memory. However, they also present some limitations, such as limited endurance and asymmetric read and write latency. Furthermore, adjusting the current hardware and software architectures to embrace these new memories in all their potential is proving to be a challenge in itself. In this paper, recent studies are analyzed to map the state-of-the-art of NVM file systems research. To achieve this goal, over 100 studies related to NVM systems were selected, analyzed, and categorized according to their topics and contributions. From the information extracted from these papers, we derive the main concerns and challenges currently being studied and discussed in the academia and industry, as well as the trends and solutions being proposed to address them.

INDEX TERMS Algorithms, design, reliability, non-volatile memory, storage-class memory, persistent memory.

I. INTRODUCTION

The increasing disparity between processor and memory performances led to the proposal of many methods to mitigate memory and storage bottlenecks [6], [107]. Recent research advances in Non-Volatile Memory (NVM) technologies may lead to a revision of the entire memory hierarchy, drastically changing computer architecture as we know today, to, for example, a radical memory-centric architecture [38]. 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 across power cycles. In summary, NVM technologies combine the advantages of both memory (DRAM) and storage (HDDs - Hard Disk Drives, SSDs - Solid State Drives).

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 [98], [99], aiming to replace the whole memory hierarchy as we know. Such change in the computer architecture would 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 swap 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

The associate editor coordinating the review of this manuscript and approving it for publication was Geng-Ming Jiang.

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 condition in order to reduce complexity and to achieve the best possible performance.

Concerned with the adaptation of current architectures to NVM devices, in the 2013 Linux Foundation Collaboration Summit [26], the session ''Preparing Linux for nonvolatile memory devices'' proposed a three-step approach to migrate today's systems to a fully NVM-aware model. In the first step, file systems will access NVM devices using NVM-aware block drivers under traditional storage APIs and file systems. This step does not explore the disruptive potential of NVM but is a simple method for making it quickly deployed and accessible by today's systems.

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 with NVM characteristics in mind, but keeping the traditional block-based file system abstraction for compatibility purposes. Significant changes in this step may include improvements such as directly mapping files into application's address space, fine tuning consistency and reliability mechanisms, such as journaling, for improved performance and eliminating processing overhead related to hard disks hardware characteristics.

The final step is the creation of byte-level I/O APIs to be used by new applications. 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. It also presents a much higher level of complexity and the requirements for such ground breaking changes are still being studied. However, at this point, applications will not only be able to take the best performance out of NVM, but will also have access to newly improved APIs and persistence models.

With this in mind, we aim to provide, with this paper, a much needed detailed overview of the state of the art of NVM-related studies, identifying trends and future directions for storage solutions, such as file systems, databases and object stores. The intention is to provide a comprehensive view of the area as well as to identify new research and development opportunities. The systematic mapping model was chosen as a method to reduce research bias and allow replication of results. Hence, we map the current state of the art of NVM storage studies including the most common issues, the biggest challenges, industry trends and standards, main tools, and frameworks or algorithms being adopted on NVM storage solutions. To that end, we provide an overview on industrial and academic efforts to push current technology to be NVM aware and more efficient. We also identify and categorize: problems related to NVM storage, what solutions to those problems have been proposed, which of those problems are still unresolved, what are the new approaches to work with NVMs and how these approaches may improve the use of NVMs. Our purpose is to gather and analyze studies related to the application of NVM technologies to different types of systems (focused on storage systems) not only to document them but also to provide a straight-forward guide for researchers unfamiliar with these concepts. Since most of the promising NVM technologies are still currently under development and access to these technologies is very limited, this paper expects to deal with more theoretical content rather than experimental results and technical applications (although those are important to the area).

Complementary to our study, a survey presented by Mittal and Vetter [94] provides an overview of NVM technologies and reviews studies focused on either combining or comparing different memory technologies. Even though the previous survey presents a classification of NVM-related solutions, it does not discuss future directions nor features that deserve additional investigation. Another study by Wu *et al.* [133] explores the challenges of adopting Phase-Change RAM (PCM) as storage, main memory or both, summarizing existing solutions and classifying them.

The remaining of this paper is organized as follows. Section II presents some basic concepts of NVM storage and technologies approached in this study. Section III details the process used in this systematic mapping study and explains the analysis and classification of the selected studies. Sections IV and V present an extensive discussion about the analysis results. In order to broaden the contribution of this paper, Section VI discusses some of the work being conducted in industry. Section VII discusses the state of the art and future directions. Section VIII presents relevant published related work. Finally, Section IX concludes this paper by providing a quick overview of the state of the art of NVM file systems.

II. BASIC CONCEPTS

This section presents some concepts that are essential to the NVM storage theme. These concepts are all intimately connected to each other and are also the base knowledge that supports the idea of NVM file systems. Further details on how these topics work together in practice will be detailed in the coming sections.

A. NON-VOLATILE MEMORY TECHNOLOGIES

The term Non-Volatile Memory (NVM) may be used to address any memory device capable of retaining state in the absence of energy. In this paper, we focus our study on emerging byte-addressable NVM technologies: memories that share many similarities to DRAM, such as low latency and byte-granularity access, while also presenting a few key distinctions, the most obvious one being their persistent nature. We also explore studies on Flash memory, as it shares some traits with byte-addressable NVM and are currently the most popular NVM in the market.

For the sake of clarity and simplicity, in this paper we make a clear distinction between byte-addressed and block-based memory devices. We refer to any byte-addressable NVM technology and memory layer as Persistent Memory (PM). For more traditional block-devices, usually based on NAND Flash memory (although PM technologies are also being employed for block storage), we adopt the term Solid State Drive (SSD).

Although these technologies have many characteristics in common, such as low latency, byte-addressability and non-volatility, they do have some key differences. These differences have direct impact over fundamental metrics, such as latency, density, endurance and even number of bits stored per cell. In this paper we cover the most popular of these technologies. There are other alternatives that are under development [72], which we do not describe here. The technologies we cover are:

- Magnetic RAM, a.k.a., Magnetoresistive RAM, 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 magnetization of the Magnetic Tunnel Junction (MTJ, a structure basically composed of two magnetic tunnels whose polarity may be alternated through the application of a magnetic field). The amplitude of the magnetic field must increase as the size of MTJ scales, which compromises MRAM scalability. MRAM presents high endurance (over 10^{15} writes) and extremely low latency (lower than DRAM) [139]. Also, although the energy necessary to read and write from/to MRAM cells is generally higher than on DRAM cells, 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 that 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 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 [72] 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 than other technologies in this list [72].
- 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 so 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 achieve around 10^7 writes lifetime [139], which may limit the usage of the technology (as main memory, for example).
- Phase-Change Random Access Memory (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 [113]. 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 [10], [113].
- 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 higher density [18] and is significantly more durable. In general, Flash memory is known for being several times slower than emerging PM technologies and usually employed solely as I/O devices, replacing magnetic disks. Despite that, Flash does share a few key characteristics with upcoming PM technologies, such as limited endurance, density, energy constraints, different speeds for read and write operations 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 [29], [54], [61]. This research notably influenced current under development PM studies. 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 PM. Furthermore, Flash memory is still currently extremely

TABLE 1. Characteristics of the NVM technologies discussed in [10], [72], [84], [106], and [113].

	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	25 μs
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

popular and it does not seem likely that SSD devices 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. However, they also have their own drawbacks that may vary from technology to technology: some have issues with endurance, some have lower performance, some do not scale well. Table 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 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 on any particular underlying NVM technology, even though we emphasize innovations and studies on byte-addressable NVM as our main interest.

B. FILE SYSTEMS

One of the simplest methods to provide NVM access 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. Hence, a handful of alternative file systems, designed specifically for NVM, were proposed, designed and implemented [25], [37], [136], [138]. NVM file systems usually take into account issues such as minimizing processor overhead, avoiding unnecessary data copies, tailoring metadata to NVM characteristics and ensuring both data protection and consistency.

BPFS [25] and PMFS [37], for example, are two early and well-known examples of NVM improved file systems, designed to provide efficient access to NVM using the 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. On the one hand, BPFS proposes the epoch barrier mechanism to reinforce ordering and to maintain consistency when writing to NVM while also avoiding cache flushes. BPFS also proposes a 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 [138]. 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.

Another common approach is the user-space file systems. Unlike the aforementioned file systems, such as BPFS that run as part of the OS kernel, user-space file systems run as regular user code through libraries and user-level processes. The usual approach is to map a region of NVM on a process address space that can be accessed by applications through programming libraries. These libraries provide to the user applications traditional (and sometimes more advanced) file system functionality without the need to interact with kernel code. This design choice aims to eliminate file system operation overhead sources related to the interaction between the process and the operating system, such as system calls and switching between protection rings. We discuss these choices and benefits further in Section V-B5.

While NVM promises to greatly improve storage performance, the fact is that, as a technology, it is still far from being mature. More traditional storage technologies such as SSD are more well developed (in both software and hardware) and suitable for larger or mass storage systems, therefore being indispensable for today's large systems. Many researchers take these characteristics into consideration and propose file

system hybrid approaches. In these approaches, the idea is to combine the best of both technologies, by, for example, using NVM to store frequently accessed data or using it for logging/journaling while using NAND Flash as back storage. These approaches may offer an array of advantages such as better scalability, improved device lifetime, increased performance, improved reliability and so on. Although the goals and methods employed by these file systems may vary, most hybrid file systems [33], [76], [105], [120], [131] face the same challenges such as predicting access patterns and managing metadata. We further discuss hybrid file system on the remaining of this paper, but most notably in Sections V-A3, V-C3 and V-C5.

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.

III. SYSTEMATIC MAPPING STUDY

This section presents the protocol used in the Systematic Mapping Study (SMS) and how the research was conducted. The SMS is based on processes used by other similar studies conducted in both Computer Science and Software Engineering [6], [64], [107]. The first step of this method is to define a protocol that presents the 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 on 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.

As explained previously, for the sake of completion, we adopt 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 memory. Therefore, we do not distinguish byte-addressable NVM from Flash in this SMS, although we do focus on the issues common to both technologies.

A. 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.

1) ESTABLISHING RESEARCH QUESTIONS

The use of research questions to guide an academic research is a very common approach. These questions help researchers to define the scope of their work, 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.

In a systematic mapping study, 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 step on 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?

2) 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 an 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) RESEARCH STRATEGY AND SEARCH STRING

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.

TABLE 2. Selected studies categorized by their contributions.

Contribution	Studies
Alternative NVM application	[4], [9], [21], [34], [41], [46], [58], [60], [59], [63], [66], [71], [74], [91], [112], [144]
Alternative software layer design	[15], [19], [24], [40], [42], [44], [48], [60], [61], [69], [73], [74], [85], [82], [92], [119], [129]
Surveys	[5], [14], [21], [35], [39], [77], [93], [94], [101], [102], [106], [111], [117], [122], [133], [137], [140]
File system design	[3], [9], [17], [25], [29], [33], [32], [37], [41], [45], [57], [58], [67], [76], [75], [99], [100], [98], [105], [104], [120], [123], [126], [130], [131], [136], [138], [144]
Novel architecture design	[3], [12], [14], [13], [11], [20], [23], [30], [33], [34], [32], [47], [49], [51], [57], [65], [76], [90], [99], [98], [97], [105], [104], [110], [112], [123], [124], [131], [132], [141], [143], [144]
Standalone technique or method	[4], [8], [13], [17], [20], [22], [19], [30], [31], [36], [43], [47], [49], [50], [51], [53], [54], [56], [58], [60], [63], [62], [69], [68], [66], [70], [71], [81], [79], [80], [78], [87], [83], [88], [86], [85], [89], [92], [91], [97], [103], [109], [112], [114], [115], [116], [118], [125], [127], [132], [134], [135], [142]

TABLE 3. 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

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 has 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 applying 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 filtered. 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 inclusion/exclusion criteria were applied over these referenced studies.

B. 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 III-A3. 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 shows the distribution of publications retrieved and selected for each search engine.

In this research, no date restrictions were specified. As a result, most of the retrieved papers were published from 2008 and 2016. Therefore, it seems that not imposing a date range was the best option in this case. We also noticed an 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) [137] and Spin-Transfer Torque RAM (STT-RAM) [93].

TABLE 4. Selected studies categorized by their area of application.

Area of Application	Studies
General purpose	[4], [5], [8], [12], [13], [15], [15], [17], [20], [19], [24], [24], [25], [33], [35], [37], [36] , [40], [42], [43], [44], [44], [45], [48], [49], [53], [56], [57], [60], [61], [59], [62], [69], [67], [66], [70], [75], [73], [79], [80], [74], [77], [88], [90], [92], [91], [93], [94], [99], [100], [98], [105], [106], [109], [111], [114], [116], [117], [120], [119], [123], [122], [126], [127], [129], [130], [131], [134], [136], [133], [137], [138], [140], [141], [143], [144]
Embedded systems	[29], [34], [71], [76], [78], [86], [101], [102], [104], [110]
Mobile systems	[22], [32], [43], [50], [65], [68], [86], [89], [115], [118], [124], [135]
Distributed systems/clusters	[9], [11], [21], [51], [58], [63], [103], [112], [142]
HPC/Scientific applications	[14], [39], [46], [51], [58], [63], [87], [132]
Data intensive applications/databases	[14], [30], [31], [41], [47], [54]
Mission critical systems	[97], [125]
Other	[3], [23], [29], [47], [65], [81], [80], [83], [85], [82]

C. 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 III-B. 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 2 classifies the studies according to the contributions they present. The idea is to identify what kind of work is currently being done in the area and the kinds of systems being designed to work with NVM. The categories listed in Table 2 were derived from the patterns identified while analyzing the retrieved studies. One of the points that the table helps to understand is that, despite the huge differences between byte-addressable NVM and disks and the emergence of NVM optimized storage [15], [24], [129], file systems are still a relatively popular area of study related to NVM. It is also important to notice that the same study may be related to more than one category. The categories of contribution identified in this work are the following:

- *Alternative software layer design:* file systems are arguably the most common way to work with persistent data, which makes them a natural target for NVM-related optimization. However, the storage stack presents other layers that may benefit from NVM fine-tuning (e.g., translation layers and block drivers). In other cases, studies explore alternatives to NVM file systems such as object-based storage systems and persistent heaps
- *Alternative NVM application:* the most common and straightforward application of emerging NVM technologies is to use it as either main memory or persistent storage. Studies in this category, however, explore the application of these same NVM technologies at other levels of the memory hierarchy, like buffers and caches. Differently from the Architecture category, these studies do not focus on NVM as storage, but as a mechanism to improve performance and reliability of other storage and RAM devices, complementing their functionalities. Also, these studies are usually driven by limitations of current NVM (e.g., density, cost and durability). Examples of alternative applications of NVM include persistent processor cache and persistent buffer for disks.
- *Surveys:* 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 many aspects of the current computer architecture.
- *File system design:* 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.
- *Novel architecture design:* 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 from (1) and (2) is either proposing a new architecture or adapting an existing one and thus belongs in 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.
- *Standalone technique or method:* this category is reserved for studies that, instead of proposing and evaluating a complete solution (e.g., a new file system or a new architecture), propose one or more mechanisms to address specific issues or explore the advantages 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.

The classification in Table 4 aims to identify the areas of application that researchers believe could most benefit

from the adoption of NVM. Overall, it helps to visualize the different ways to adopt NVM and their impacts on the areas of application and to understand the motivation behind the solutions proposed by the selected studies. The results in Table 4, however, show poor distribution, leaning strongly towards the ''general purpose'' category, as studies seem to not pick a single specific area of application to focus on. This shows that currently most NVM-related works are interested in addressing NVM issues and challenges in general and exploring the impacts of these technologies on a more generic and abstract structure rather than on specific applications and environments.

The solutions that are not categorized as general purpose may be divided in basically 6 categories: mobile, embedded, distributed, HPC, Data Intensive and mission critical systems. These are areas of application where storage constraints are among their main challenges, thus the motivation for employing NVM to address the storage issues. All areas of application are characterized as the following:

- *General purpose*: these are studies that 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.
- *Embedded systems*: embedded systems may benefit greatly from NVMs' energy efficiency, resistance to impact (no moving parts), and small size. Studies focused on embedded systems may cover a large array of applications and architectures, like wireless sensor networks and consumer electronics. Additionally, these studies usually assume architectural and technological constraints such as limited capacity and high cost for NVM chips. Therefore, embedded system-related works are characterized by their efforts to save physical memory by avoiding data duplication and providing compression.
- *Mobile 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.
- *Distributed systems/clusters*: studies of 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 these 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.
- *HPC/Scientific applications*: high-performance computing (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/databases*: 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.
- *Mission critical systems*: these 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.
- *Other*: other applications include sensor networks [29], transactional systems [47], highly concurrent systems [85], semantic storage and application [3], high reliability (through data redundancy) systems [81], virtualized systems [82], streaming systems [65] and multidimensional storage [83].

Our last classification focuses on the problems addressed by each publication. The idea 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 paper. Figure 1 shows how this classification relates to the previously presented ones. For each topic of NVM (represented by the vertical axis), it shows what contributions and target environments (the horizontal axis) are most commonly associated with it. The bubbles represent the number of studies that each category cover.

We discuss each of these topics extensively in Sections IV and V, since understanding these challenges is essential for researchers interested in NVM. We believe this discussion to be the main contribution of this work. In this paper we separate topics on NVM between storage/file system specific topics and main memory/general topics. For the purpose of this distinction, storage is considered whenever a structure, like file system or object system, is built over NVM and used to consistently store long term data and metadata (Section IV is dedicated to these topics). Thus, cases where NVM is used as operating memory, similar to current volatile memory, or used as buffer or cache, are not considered examples of storage. Hence, in Section V we discuss topics that are not storage exclusive: they are also relevant when NVM is used for other purposes like main memory or write buffer.

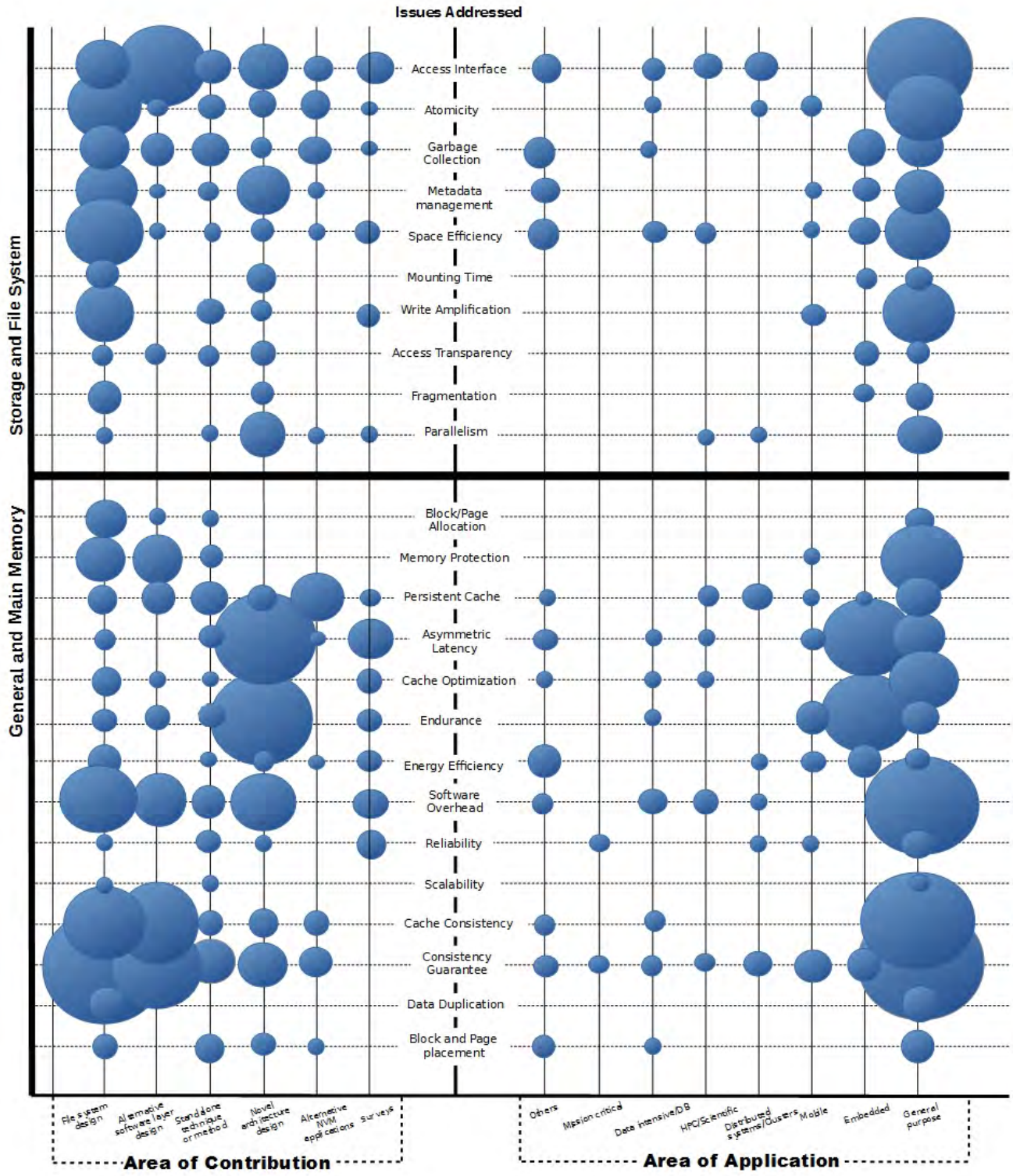


FIGURE 1. Bubble plot illustrating the focus and distribution of NVM research: the X axis illustrates the types of contribution (left) and area of application (right) previously presented in Tables 2 and 4 respectively. The Y axis represents the issues addressed, separated between Storage specific and generic/main memory (Tables 5 and 7).

IV. STORAGE AND FILE SYSTEMS

In this section, we start presenting the results of our analysis and classification of the selected studies. We start discussing topics that are intimately related to emerging PM file systems and NVM storage systems in general. These are common subjects, challenges and concerns that arise when designing

TABLE 5. Topics closely related to storage and file systems on NVM and the studies that explore them.

	Problem	Studies
Functional	*Access Interface*	[3], [5], [14], [15], [20], [24], [40], [44], [48], [57], [58], [63], [62], [90], [92], [94], [99], [100], [98], [103], [106], [123], [126], [127], [129], [136], [144]
	Atomicity	[17], [20], [25], [41], [42], [43], [91], [103], [122], [127], [138], [144]
	Garbage Collection	[29], [30], [32],[60] [69], [89], [101] , [104], [136], [138]
	Metadata Management	[3], [8] , [17], [20], [22], [33], [32], [48], [57], [61], [62], [76], [74], [83], [89], [105], [104], [118], [119], [123], [130], [131], [138]
Non-Functional (Quantitative)	*Space Efficiency*	[14], [25], [29], [31], [53], [61], [74], [89], [102], [109], [126], [137]
	Mounting Time	[57],[105],[104], [138]
	Write Amplification	[20], [25], [50], [54], [68], [92], [94], [79], [87], [111]
Non-Functional (Qualitative)	*Transparency*	[19], [61], [76], [141]
	Fragmentation	[76], [85], [136]
	Parallelism	[12], [58], [85], [90], [122], [137]

a storage based on the NVM technologies presented previously. Many of the topics discussed here are traditional issues rooted in the very concept of file systems, such as fragmentation, metadata and space efficiency. Others are more closely related to the NVM technology characteristics, such as access interface optimization and garbage collection. We further classify these topics according to their nature as either functional or non-functional issues. At the end of each of these subsections, we summarize the main points and complementary insights provided by the reviewed studies. The topics discussed and the studies that compose them are presented in Table 5.

A. FUNCTIONAL

This section discusses functional topics and requirements of NVM-based storage. In this paper, we considered ''functional'', topics related to basic storage functionality, behavior and operations exposed to applications. Studies in this category cover a variety of subjects, including operations such as defining transactions, manipulating metadata and allocating/mapping NVM address ranges.

1) ACCESS INTERFACE

Due to PM byte-addressability, an issue that is extensively discussed in several papers concerns the method of accessing and managing these memories. At the hardware level, the most common question is whether PM should be accessed as a block device, like disks and SSDs are, or through a memory-bus with byte-addressable granularity. On the one hand, using the block-driven approach offers transparent access to PM with relatively little modification to the existing software stack and robustness in terms of error detection and memory protection. On the other hand, accessing PM with a byte-granularity interface may lead to an unprecedented high performance, bringing data closer to the processor and fully exploring PM potential. At the software level, the discussion is whether data in PM should be accessed through a more traditional file system API, through an interface closer to that of today's main memory (allocating persistent regions and data structures) or using a novel, more application-friendly interface, such as heap-based and key-value interfaces [48], [61]. 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 [40], [129].

A good and simple example of PM file system design is PMFS [37]. PMFS is designed specifically to avoid well-known sources of file system overhead on PM. It provides traditional file system interface and behavior such as read, write and file mapping operations, while offering better performance than disk file systems by eliminating expensive page copies and processing overhead. It also allows applications to map files directly to their address space as memory arrays through the XIP-enabled (eXecute In Place) *mmap* implementation. In general, this traditional file system approach on PM is essential for applications to migrate to PM transparently and to support legacy systems, but it also offers common file system advantages such as hierarchical namespace, access control and system interoperability.

NV-Heaps [24] proposes a persistent object store based on PM. NV-Heaps is a very simple and lightweight tool focused on high performance that provides an easy-to-use set of primitives including declaration of persistent pointers, persistent data structures (e.g., lists, trees and maps), starting atomic transactions, inserting and retrieving objects from the store. Unlike file systems, key-value object stores are organized in a flat namespace (no directories) where objects are retrieved simply through their key instead of a directory path. Today's key-value stores are already moving into RAM and present well-known advantages over file systems such as scalability, ACID (Atomicity, Consistency, Integrity and Durability) transaction semantics and efficient indexing.

Mnemosyne [129] grants users access to NVM by persistent regions. Persistent regions are segments of memory that can be manipulated like regular volatile memory by user code but are allocated in PM. In order to allocate these persistent regions, users can either mark a variable as *pstatic* or calling the *pmap* function. These persistent regions are

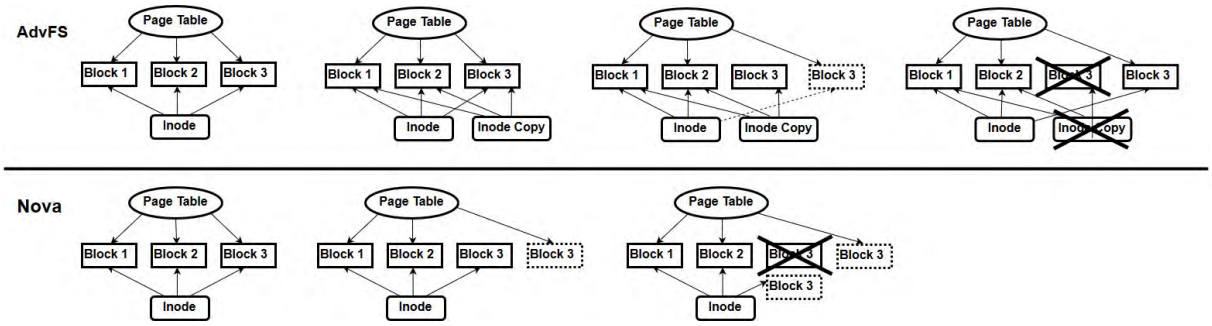


FIGURE 2. Atomic file mapping implementation as described by Nova [138] and AdvFS [127].

stored in files (that may be kept in a secondary storage, like an SSD) and mapped (through *mmap* system call) on demand to the NVM when their owner process is started. Additionally, Mnemosyne offers users 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 properties of all changes made to persistent data inside this block of code through a transaction structure.

2) ATOMICITY

Atomicity mechanisms enforce data consistency by ensuring that operations modifying data are either successfully completed or completely ignored (rollback). However, atomic transactions in NVM face particular challenges, especially concerning efficiency and scalability, due to the larger impact of transactions overhead and write amplification issues represent on faster memories. Hence, many studies [25], [48], [61], [73], [127], [138] propose different methods to improve the performance of these mechanisms while also enforcing the consistency and security of the storage system.

The Byte-addressable Persistent memory File System (BPFS) [25] 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, updates the necessary data and writes these copies in new blocks. The main drawbacks of this solution are 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.

A file system mechanism that is particularly susceptible to consistency problems is the ''file mapping'' also sometimes called ''memory mapping''. File mapping is a popular method of accessing data due to its flexibility, efficiency and its similarity with regular volatile memory allocation methods. These characteristics led file mappings to be frequently employed as the base for PM-based storage systems such as Mnemosyne [129]. While traditional mapping mechanisms require data to be copied back and forth from the disk to DRAM, PM file systems like PMFS allow applications to access PM 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 in file data. With that in mind, a few studies propose making updates to these mappings atomic [103], [127], [138]. Atomic file mapping solutions differ from traditional mappings by: preventing OS from automatically writing back updated blocks and performing updates atomically during *fsync/msync* system calls using out-of-place updates. Figure 2 depicts how these atomic updates behave. 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 these mappings are made durable (through *fsync* or *msync* calls, for instance).

3) GARBAGE COLLECTION

Out-of-place updates, commonly performed by wear-leveling and atomicity techniques, mean that updates are actually written in a new block and the old version of the updated data is marked as invalid. The garbage collector is the one responsible for identifying blocks marked as invalid to allocate space for new data. This may or may not involve additional writes. Although this process is usually executed asynchronously (ideally when the storage device is idle), it has significant impact on the storage overall performance and energy consumption, thus demanding smart optimization [104].

ELF [29] 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 may also merge partially valid blocks (i.e., blocks that contain some pages marked as invalid) to free additional space.

DFTL (Demand-based Flash Translation Layer) [69] 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 [138] 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.

4) METADATA MANAGEMENT

A critical point of optimization in file systems is related to metadata structure and management. It is known that access to metadata is intensive [20] and updates are very frequent. Furthermore, corruption in metadata may lead to the corruption of big portions or the entirety of a 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 storage. Optimizing metadata for NVM file systems, for instance, must consider many aspects, such as inefficiency of write operations, limited endurance and byte granularity. We also include in this section studies [33], [76], [105] that suggest the use of PM as a metadata-only storage mixed with SSD in a hybrid architecture, since they consider details like the high cost-per-byte of pure PM devices in the near future and their byte-granularity while designing their structures.

LiFS [3] also exploits the advantages of PM, in this case to expand the capabilities of metadata 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 key-value attributes implemented through hash tables and linked lists. This additional information may be queried by operating systems and applications for a variety of purposes including indexing, semantics analysis and file history tracking.

The Aerie [130] file system architecture exposes PM to user-mode programs so they can access files without kernel interaction. By accessing file system data and metadata directly from user-mode, Aerie file system can optimize interface semantics and metadata operations to address specific needs of the targeted class of applications. The implementation of the file system as a user-mode library can also avoid the costs of calling into the kernel due to changing mode and cache pollution.

5) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Most storage solutions in the literature present some form of atomic operation. In PM, where the storage is exposed to the CPU and it is more error prone compared to disks, atomic operations must take in considerations things like hardware errors, data retention and reordering (e.g., in processor cache and memory controller). These operations also seek to provide better concurrency using data replication, relaxed ordering constraints, per CPU metadata and lock-free data structures. Using light fine-grained (subpage) versions of traditional techniques such as shadow paging and redo logging is also a popular alternative.

Regarding NVM access interface, at the OS level, most studies deal with NVM-optimized object stores, file systems and block drivers as the means to securely access NVM. Additionally, novel PM OS layers like the Persistent Memory Block Driver (PMBD) [19] help providing legacy application and traditional file systems access to PM. However, POSIX compliant file systems have a few inherent issues, such as the cost of changing modes and cache pollution caused by entering the OS code, that may significantly limit the performance of accessing PM. On the application level, following Mnemosyne, several works [8], [15], [24], [44] propose higher-level NVM programming libraries, usually implemented over a PM mapped area. User-level access to PM may eliminate overheads related to OS storage layers but makes implementing security, protection and interoperability between systems challenging. Table 6 shows the syntax of three of these libraries and their similarities. Internally, these libraries require an NVM-aware file system allowing direct access to files in PM via memory-mapping.

Metadata studies have basically three goals. First is adopting lightweight and byte-granularity structures commonly in the form of B-trees and hash maps (inodes and inode tables). Next is making storage structures more concurrency-friendly. Finally, solutions may also target improvements specific for memory-level storage such as contiguous address spaces to optimize integration with page tables and the TLB.

B. NON-FUNCTIONAL - QUANTITATIVE

In contrast to the topics presented in Section IV-A, we classify the remaining studies as non-functional. In this context, ''non-functional'' describes system characteristics (e.g., performance, reliability, scalability) rather than its behavior. For further characterization, we separate non-functional topics in two subcategories. The first one is discussed in this section, and targets subjects where improvements can be measured and estimated, usually studies focused on optimizing performance, energy consumption or device capacity usage. The second is discussed in Section IV-C.

1) MOUNTING TIME

One of the problems addressed by traditional SSD file systems concerns the file system mounting performance. Basically, the mounting process involves finding and reading file system metadata structures (such as block tables and superblocks) to build OS internal structures. In some cases, e.g., mobile phones and digital cameras, the performance of this process may be critical for the system's usability. On the one hand, since NVM technologies have endurance limitations, keeping frequently updated data (such as superblocks and inodes) in a fixed position in the device may not be a good idea. On the other hand, since NVM 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 [57].

An efficient and simple way to deal with the mounting time constraint is proposed by FRASH [57] and PFFS [105]. Both are systems designed for hybrid storage models that use PM

TABLE 6. Code example for some of the main solutions on PM API: atomically adding an item to a persistent list.

NV-Heaps

```
class List : public NVObject {
  //Macros generate NV-Heaps persistent pointers and atomic access methods
  DECLARE_POINTER_TYPES(List);
public:
  DECLARE_PTR_MEMBER(NVList::NVPtr, next);
};
void persist()
{
  NVHeap *nv = NVHOpen("default.nvheap");
  NVList::VPtr root = nv->GetRoot<NVList::NVPtr>();
  AtomicBegin {
      while(root.get_next() != NULL) root = root.get_next();
      //Insert new item - NV-Heaps makes it persistent transparently
      root->set_next(new List());
   } AtomicEnd
}
```

Mnemosyne

```
struct list {
        list *next;
}

void persist(list *list_head){
  list *new_item;
  atomic {
    //Allocate a persistent region
    new_item = pmalloc(&list, sizeof(*list));
    list_head->next = new_item;
  }
}
```

SoftPM

```
struct list_item {
        ...
}
struct rootp {
        list_item *head;
} *list;

void persist(){
  list_item *list_head;
  //Allocate a persistent container to store the list
  root = pCAlloc(&list, sizeof(*list));
  list_head = malloc(sizeof(*list_item));
  list->head = list_head;
  //Persists the container and the list
  pPoint(root);
}
```

for metadata and indexes and SSD for general storage. This approach has many advantages: the scanned area is smaller (the amount of PM needed in this approach is just a small fraction of the full size of the SSD), the scan for metadata is faster due to PM low latency (compared to SSD) and memory copies can be reduced, since metadata can be directly accessed in PM.

2) SPACE EFFICIENCY

High capacity is among the first and most desirable characteristics one might expect from a computer's storage. File systems and techniques that focus on space allocation in storage are usually aimed for systems with limited physical storage, like mobile and embedded systems [29], [71]. The most common methods of reducing space usage are compressing data and simplifying metadata structures [31], [61], [126] such as Muninn's bloom filters that are designed to reduce the amount of stored metadata.

MRAMFS [126] is a file system designed to be space efficient. It compresses file data as well as metadata, trying to balance the trade-off between access performance and compression level. A different approach is given by the dynamic over-provisioning technique [53], which focuses on efficiently using the additional space inside SSD to allow

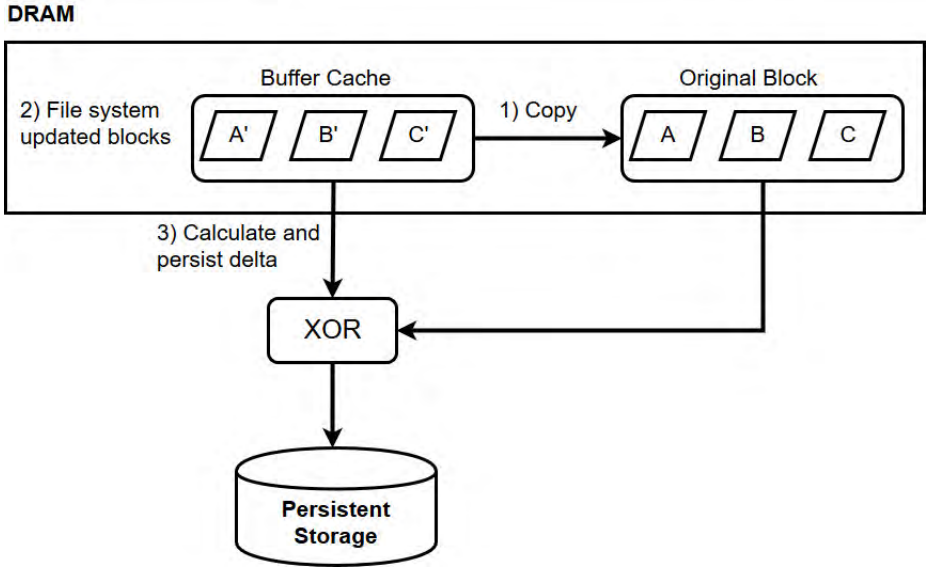


FIGURE 3. MRAMFS method to capture the delta between original and modified versions of a block [68].

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. NVM Compression [31] is a hybrid technique that combines application-level compression with specialized Flash Translation Layer primitives to efficiently perform block management tasks required by compression.

3) WRITE AMPLIFICATION

Besides the atomicity solutions presented previously, other NVM storage mechanisms like wear-leveling (Section V-B3), shadow-paging (Section V-C2) and metadata updates in general may introduce additional writes meaning that writing a chunk of data to persistent storage may have side-effects and require updating or moving other data blocks or metadata around. In the file systems area, this phenomenon is commonly known as write amplification. Due to its performance impact, minimizing the write amplification is one of the most common goals of novel wear-leveling and consistency techniques. Thus, most of the studies that tackle this particular challenge are also intimately related to atomicity, consistency and endurance improvement techniques.

OFTL (Object-based Flash Translation Layer) [92] uses page level metadata and reverse indexes 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, which are among the main sources of write amplification in traditional systems. To reduce the effort required 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 [20] 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 of 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 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 [68] 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. When a write is issued (e.g., through a system call), it captures the difference between original and modified block and calculate the compression rate (Figure 3). It then stores the compressed difference in NVM so it can be securely flushed back to the long-term storage (SSD).

4) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Efforts towards reducing write amplification is more commonly seen on SSD where metadata structures are block-based and the physical device's blocks must be erased before they can be rewritten. These factors contribute to the overhead of updating user data. Write amplification on PM on

the other hand usually works with fine-grained metadata and logs to minimize replication. Fine-grained and shared metadata structures may also benefit storage's space efficiency. Additional ways to improve space usage on small capacity devices may include using PM for metadata and for hot data only.

C. NON-FUNCTIONAL - QUALITATIVE

Similar to Section IV-B, this section presents topics related to storage system's non-functional studies. The difference in this case is that qualitative topics, due to their nature, are not measurable and thus their analysis follows descriptive approaches to evaluate correctness and effectiveness.

1) FRAGMENTATION

In addition to the increased per-operation latency, write amplification and out-of-place updates may also introduce fragmentation issues. Although NVM present homogeneous latency for both random and sequential access, fragmentation may still become a challenge in some cases, specially in SSD. For instance, scattering multiple pages of a file through different erase blocks in SSD may cause an increase in the number of block merges [76] and increase garbage collection overhead (see Section IV-A3). In other cases, dealing with larger page sizes (larger than the traditional 4 KB), or even segments, may cause problems of internal fragmentation [136].

Flash Translation Layer is a software layer between the file system and SSD that implements address redirection to provide support such as wear-leveling. The hybrid FTL (Flash Translation Layer) design in [76] employs PM to store critical storage-related metadata, such as allocation bitmaps and page map tables. Furthermore, the solution uses virtual blocks to keep track of logically related pages and their physical addresses. The FTL then uses this information to avoid placing logically related pages in different erase blocks, prioritizing keeping related pages contiguously and clustering large writes into contiguous physical address ranges. Also using PM the fragmentation impact of the metadata itself is minimized, since updates may be made in-place without block merges.

When discussing fragmentation in NVM, most studies focus on the external fragmentation. The adoption of superpages in SCMFS [136] (PMFS also implements huge pages) brings with it an example of internal fragmentation. Internal fragmentation happens when a block-based storage allocates a larger chunk of physical NVM than it actually needs. In this particular case, regular 4 KB pages are used for fine-grained data, while large pages (2 MB) are used for bigger datasets. While for traditional 4-16 KB blocks it may not represent a major issue, with huge pages that may be as large as 2GB, it may cause significant waste of physical memory. SCMFS adopts the 2 MB superpages to improve the hit ratio on TLB and speed up the address look-up process. Although it benefits the performance of the address translation process, this approach may present issues regarding paging complexity (e.g., mechanisms to avoid fragmentation), moving and copying large pages (e.g., the overhead of moving an entire page to access a single address is much larger now) and may create internal fragmentation issues if not properly handled.

2) PARALLELISM

The growing parallelism in SSD architectures is rapidly increasing, greatly improving the throughput of NVM-based block devices. Additionally, everything from modern operating systems to programming models have been transitioning towards a more concurrent environment. However, traditional file systems usually fail to take full advantage of such parallelism wasting valuable resources [137]. PASS [85] is a scheduler developed to take advantage of modern SSD architecture and its parallelism. PASS divides the storage in logical scheduling units, avoiding mutual interference caused by concurrent operations and allowing multiple channels of data at the same time.

3) TRANSPARENCY

Although the integration of PM technologies in the memory hierarchy may lead to ground-breaking developments and paradigms switches in computer architecture, many applications may greatly benefit from these technologies as simple high-speed block devices. With this in mind, some 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.

In order to provide transparency and a rich interface to access PM, Muninn [61] presents an object-based storage model. In this approach, data and metadata management are delegated to an object-based storage device (analogous to a block device), which 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, in order 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 semantics and calling the appropriate object-based device operation.

4) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Studies on external fragmentation issues related to PM are practically inexistent, as PM does not suffer from the limitations of SSDs, such as the need to erase before rewriting blocks. In SSD, fragmentation solutions are already relatively mature and implemented on most commercial devices, usually employing block remapping and write buffering on the FTL. On the other hand, internal fragmentation of PM is more of a potential issue. There is a tendency on exploring more

TABLE 7. Topics that are not exclusive to file system or storage.

Category	Problem	Studies
Functional	*Block/Page Allocation*	[8], [69], [99], [100], [98], [111], [118], [136], [133], [138]
	Memory Protection	[19], [37], [43], [100], [98], [129], [136], [138]
	Persistent Cache	[4], [9], [21], [34], [46], [51], [60], [59], [65], [66], [71], [74], [77], [82], [91], [133], [137], [144]
Non-Functional (Quantitative)	*Asymmetric Latency*	[4], [30], [49], [54], [71], [80], [78], [83], [90] , [93], [94], [104], [110], [124], [132], [133], [137], [141]
	Cache Optimization	[23], [25], [30], [69], [77], [109], [111], [132], [140]
	Endurance	[17], [30], [50], [54], [61], [83], [92], [94], [101], [102], [105], [104], [110], [111], [124], [134], [135], [133], [137], [141]
	Energy Efficiency	[4], [11], [29], [45], [65], [71], [86], [94], [106], [133] , [137]
	Software Overhead	[8], [12], [14], [13], [11], [15], [24], [25], [35], [44], [47], [48], [62], [99], [111], [117], [120], [119], [123], [130], [136], [138], [140], [141]
	Reliability	[5], [43], [75], [88], [94], [97], [111], [114], [125], [142]
	Scalability	[62], [109], [138]
Non-Functional (Qualitative)	*Cache Consistency*	[15], [24], [25], [37], [42], [44], [47], [56], [69], [67], [70], [129], [133], [144]
	Consistency Guarantee	[5], [15], [20], [19], [24], [25], [29], [32], [37], [41], [43], [44], [48], [49], [54], [57], [61], [63], [68], [66], [81], [75], [73], [79], [87], [92], [91], [94], [97], [102], [103], [111], [114], [115], [120], [119], [123], [124], [125], [127], [129], [131], [136], [133], [138], [143], [144]
	Data Duplication	[37], [67], [98], [100], [120], [119]
	Data Placement	[30], [36], [51], [62], [80], [78], [112], [116], [131], [141], [143]

flexible addressing models such as segmentation and large pages to make addressing large PM ranges easier, but the pitfalls of these methods, including internal fragmentation, still demand study (more in Section VII-Q). Nevertheless, the support for multiple page sizes is already implemented in the Linux kernel and most PM file systems [37], [136].

While parallelism in SSD is more commonly discussed in the literature, in PM it is not much explored. Parallelism may be improved on PM on the memory controller level by, for example, smart write scheduling or on the system level with per processor metadata [138].

V. MAIN MEMORY AND GENERAL NVM

In this section we focus on issues that are not limited to storage systems but may also affect other applications of NVM, such as PM-based main memory or persistent buffers. It is important to mention that, to some degree, all these topics are also relevant to storage systems. The idea is to separate storage related issues from more generic matters that may apply to most diverse applications of emerging NVM technologies. A list of common problems addressed by studies in the area are shown in Table 7. In this section these problems and their impact over the system are detailed. This discussion is an extension of the one presented in Section IV and therefore follows the same structure.

Finally, Figure 4 depicts how the topics presented here and in Section IV relate to the applications and OS structures in today's systems. For each topic discussed in this survey, the diagram intends to illustrate what component of the architecture it is more closely related to and at what level it is more naturally addressed, according to the reviewed studies. It is worth noting that the same challenges may be addressed at multiple levels: for instance, consistency and reliability mechanisms may be (and usually are) implemented at the device driver level or they could be implemented by the file system itself or even in a user level library (which could be a user-level file system).

A. FUNCTIONAL

As discussed in Section IV-A, functional topics are related to NVM software behavior and operations. Thus this section explores challenges and requirements related to functionalities such as memory allocation and memory protection.

1) BLOCK/PAGE ALLOCATION

Allocating blocks is one of the most common operations of a file system and storage in general. In traditional file systems, optimizing this process to reduce fragmentation and to optimize disk access is critical and, thus, it carries a high responsibility. However, as we further discuss in Section V-B5, such an effort is unnecessary and represents a bad practice in the use of the NVM technology. Furthermore, NVM file system's block allocation introduces new concerns such as dealing with the devices limited lifetime and with concurrency. Studies in this category aim to optimize block allocation algorithms and policies to reduce its overhead and explore the characteristics of both NVM and the in-memory file system architecture.

SCMFS [136], for example, uses space pre-allocation, allocating blocks ahead within 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

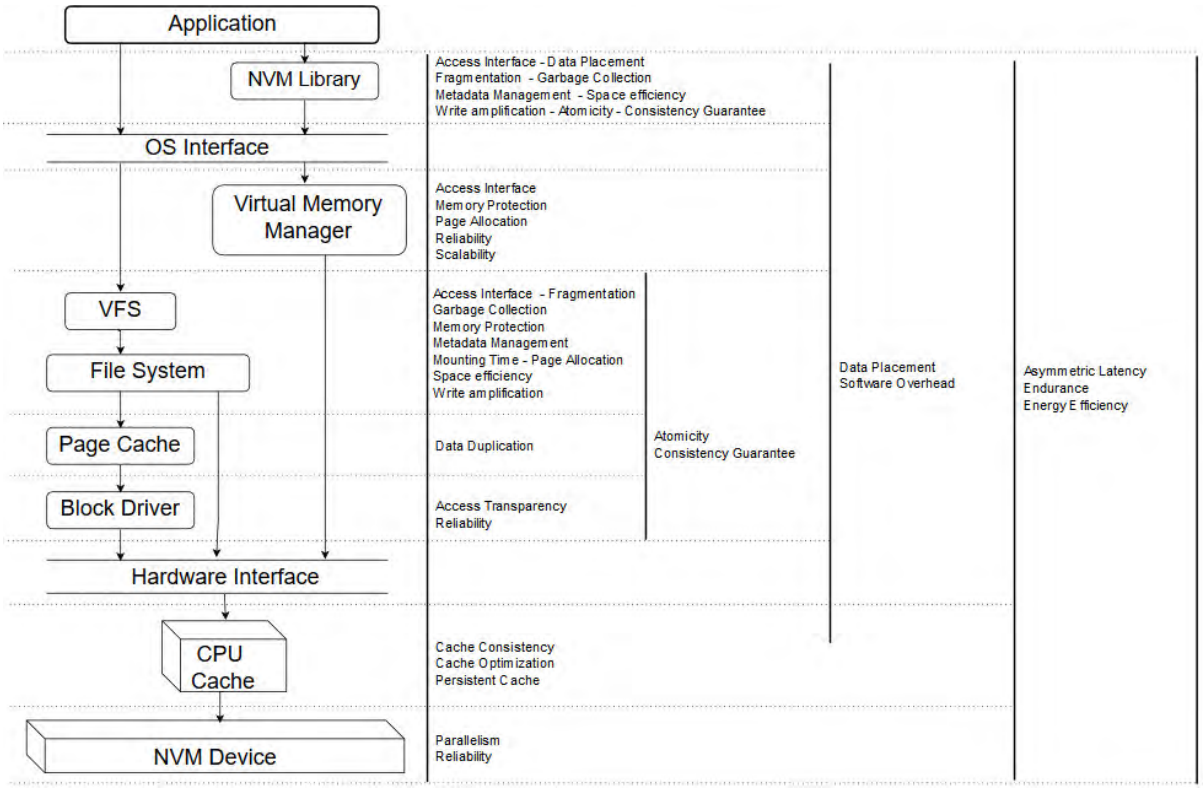


FIGURE 4. Diagram connecting the challenges identified with the respective architecture levels that address them.

appends, easier. The W-Buddy [118] algorithm, organizes free memory space in a structure similar to a binary tree, and uses allocation bitmaps and update counters to find the best fit when allocating new chunks of memory. The binary search is significantly faster than a sequential search, reducing the time necessary to find an ideal page to allocate.

2) MEMORY PROTECTION

Stray writes (e.g., writes performed on an invalid pointer, possibly referencing memory out of the process' address space) is a software issue that may be the result of bugs in the kernel or in system's drivers, and represent a serious threat to data integrity [19]. If the data in question is stored in PM, it may be permanently corrupted. The mechanism adopted to avoid improper access to memory addresses by a process is commonly known as memory protection. Some studies propose different methods of memory protection designed for PM to avoid improper access to non-volatile memory pages [19], [37]. These methods may be implemented in software or use specific hardware to enforce its policies.

A simple and common method to avoid protection issues 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 [43] 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 [37] does something similar to ensure memory protection, avoiding corruption caused by stray writes (see Listing 1). The main issues with this approach are that it still exposes memory to stray writes (only in small windows of time) and that changing the read-write bits of page table entries requires flushing the TLB, which is a very expensive process.

```
void *xmem = pmfs_get_block(superblock,
    block_address);

size_t offset = pos & (superblock->blocksize - 1);

//Enables writes to the file system
pmfs_xip_mem_protect(xmem + offset, count, 1);

//Writes data to block
__copy_from_user_inatomic_nocache(xmem + offset,
    buf, count);

//Disables writes to the file system
pmfs_xip_mem_protect(xmem + offset, count, 0);
```

Listing. 1. PMFS code to write a block.

A more robust solution comes in the form of the Memory Protection Keys (MPK) proposed by Intel (see Section VI-B). This mechanism uses a set of new registers added to the processor to 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 the TLB flush issue. In terms of process, the steps of protecting memory using MPK should be similar to those performed by EVENODD and PMFS: the system protects its data using MPK, allowing writing to it only during a small time window (e.g., during a file system *write* call).

3) PERSISTENT CACHE

Given the challenges of integrating volatile caches with persistent storage layers, some studies [9], [59], [71], [82], [144] suggest the use of PM for caches and buffers. Since PM (and NVM in general) retains data across power cycles, PM buffers can be very useful in improving storage reliability and security, while reducing the need for periodic flushes to long-term storage. However, at their current state PM is not a feasible replacement for SRAM on processor-level cache due to endurance and latency limitations. The reflection of this is that few papers actually explore the idea of persistent processor cache. Thus, the impacts of a persistent cache at the top of the memory hierarchy and the requirements for appropriate policies (e.g., locality, eviction and data placement) for such cache are still unclear. Thus, studies that explore PM-based cache management usually present metrics and policies to take full advantage of these technologies and to predict access patterns and improve their hit ratio while also studying their impact in today's systems (e.g., impact of cache retention).

TxCache [91] 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 on 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 (on 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 in Kiln [144] has similarities to TxCache transactions, however, modified to work with last level processor cache policies instead of disk caches. Data is transferred from volatile to non-volatile cache in a transactional 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: PM-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.

Mittal and Vetter [95] proposed an algorithm to cope with PM limited lifetime on cache level. This algorithm keeps a counter for each cache block, incrementing it for every write received. Once the counter reaches a predetermined threshold, the cache block is written back to memory and invalidated. Also, its LRU (Least Recent Used) counter is not updated. With this, it ensures that the hot data stored in that cache block will be reassigned to a cold block due to the LRU algorithm.

4) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Like with most mechanisms in NVM systems, the main requirement of an NVM block/page allocator is for it to be simple and lightweight. Also, to cope with a high allocation demand, concurrency-friendly structures are also necessary [138]. With this in mind, many solutions seek inspiration from traditional VM allocators rather than file system block allocation methods. Other factors may also come to influence NVM allocation such as the number of times a physical block has been overwritten (endurance, see Section V-B3).

Persistent cache and buffers may bring many benefits for computing systems such as energy efficiency and fault-tolerance. There are two main challenges to the adoption of PM as cache. The first one is performing writeback to the long-term storage atomically with minimum performance impact. Traditional cache policies are not designed to deal with transactions and persistence at memory-level. The second is that to be used as processor cache, PM requires high endurance. However, even if these technologies can endure enough write cycles, there is still the fact that cache is very locality sensitive and overwriting the same cache line multiple times is in its very purpose. In this case, neither traditional cache policies nor storage and main memory wear-leveling solutions are properly fit to optimize persistent cache lifetime, thus requiring specific solutions.

B. NON-FUNCTIONAL - QUANTITATIVE

In this section we present topics related to quantifiable non-functional NVM system requirements such as minimizing power usage and processor overhead.

1) ASYMMETRIC LATENCY

Another common property shared by NVM technologies is the asymmetry in the latency of their operations [4], [49], [54], [83]. In these devices, write and erase operations are much slower than read operations. Furthermore, in SSD, data cannot be updated in-place as blocks need to be erased before they are overwritten, which aggravates the problem.

The impact of such slow write/erase operations may become prohibitive to the adoption of NVM as either storage or main memory, and strategies must be adopted to either reduce the amount of writes that reach NVM (usually software level) or to reduce the latency of these operations (in hardware). This property is a key factor in the design of most NVM systems.

A common way to deal with the high cost of writes is by simply employing 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 PM) [110], techniques like lazy and fine-grained writes could reduce the amount of writes to reach the PM device in order to reduce the impact of asymmetric latency and to improve its lifetime. In this case, DRAM acts like a traditional page cache and pages are only written back to PM when dirty during the eviction process executed by page replacement policies.

Another approach, named i-PCM [4], explores the fact that the amount of energy and time needed to make a write to PM is directly proportional to the duration of its persistence. This means that higher latency and energy dissipation are needed to make data durable in PM for longer periods. However, some pages may be updated very frequently (e.g., file system metadata) while others may belong to temporary files. In these cases, the persistence process may be relaxed, reducing the durability of data but also the latency of write operations. In one hand, i-PCM can distinguish hot and cold files and apply different write intensities for each one, which, since hot files usually dominate the accesses to storage, may improve performance as well as energy efficiency significantly. On the other hand, PM may need to be periodically refreshed whenever the hot-cold prediction fails in order to not lose data, a process that may degrade lifetime and increase energy consumption.

Another interesting method to reduce write latency is the read-before-write technique [86]. 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.

2) CACHE OPTIMIZATION

While the studies presented in Section V-C1 focus on providing cache and buffer consistency, others focus on tailoring buffer management and cache policies to NVM storage and main memory. These studies seek the optimization of cache and buffers using NVM-aware algorithms. For example, traditional policies for disk buffers may expect random access to be much slower than sequential access which is not true for NVM. Furthermore, block-oriented write buffers may not take full advantage of PM byte granularity that could be used to implement more complex fine-grained update mechanisms in order to improve write performance [37], [49]. Thus, NVM-aware policies and techniques are needed to make these intermediary memory layers smarter and more efficient.

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 to organize cache lines in epochs. The epoch barriers are issued through an epoch instruction, just like with *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. In the case of epoch barriers, however, each cache line is marked as belonging to a specific epoch and each epoch has a precedence (through a sequential number, for example). In this case, if epoch A precedes epoch B, the cache lines belonging to epoch B can only be evicted if every cache line from epoch A has already been evicted. Thus, cache is not flushed and data is only written to persistent storage when evicted, potentially improving cache usage and hit ratio. Another way to reduce flushes to memory is using a non-volatile processor cache: since the PM 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 [144].

3) ENDURANCE

As mentioned before, the limited endurance of NVM devices may cause some cells or blocks/pages to wear out faster than others, since some data may be updated much more frequently than others [61], [118], [135]. This results in permanent loss of these blocks, reduced storage capacity and possibly in file corruption. While the level of endurance required of an NVM depends greatly on its position in the memory hierarchy, it is well known that the more you can safely write to an NVM device without risking hardware failures the better.

W-Buddy [118] is a wear-aware memory allocator that extends the Buddy [55] 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, 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.

Wu *et al.* [134], [135] use two different heuristics to identify if frequently updated (hot) files are present. 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 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. These heuristics are applied to the Android operating system and extensively studied by Wu *et al*. [135], that explores file organization and application-related metadata to identify hot and cold files. This information is used to store hot data in physical blocks with low write counters and cold data in more worn-out blocks, interchanging them during the system life-cycle as the blocks write counters increase. It should be noted that to achieve higher lifetime in a hybrid memory architecture, hot data could be also stored in a more resilient memory device (e.g., DRAM) while cold data is kept in the device with lower endurance.

4) ENERGY EFFICIENCY

In general, the concern about energy consumption in computing has grown significantly, particularly in the past few years, as the technology continuously allows machines to scale to unprecedented proportions. In these cases, energy consumption may become a limiting factor in designing more powerful architectures and is a key factor to determine aspects such as maintenance cost. Both main memory and storage are among the main sources of energy consumption and therefore are good candidates for optimization. On the other hand, embedded and mobile systems based on NVM, with limited or no access to long-term power source may also benefit from a smarter more efficient usage of power [86].

Many solutions for energy efficient usage of NVM are closely related to those presented in Section V-B1, since reducing expensive write operations to NVM is the key to reduce overall memory energy consumption [45], [65], [137]. Thus, approaches such as using volatile buffers are common solutions to both latency and energy optimization, even though volatile memories like DRAM and SRAM usually consume more energy than PM while they are not being accessed. In another method [71], PM is used as a cache for instructions only, in order to leverage performance while also improving the persistent memory's lifetime by storing only read-intensive data (in this case, instructions) in PM. PM is a good candidate to store read-intensive data as reading is cheaper in terms of both energy and time and also because PM do not need periodic refreshes like DRAM. Copy-before-write can also be used to change the minimum number of bits in a page [86]. The algorithm is based on 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.

5) SOFTWARE OVERHEAD

Since, in magnetic disks, sequential writes and reads are much faster than random operations, great effort is made by the file system to avoid fragmentation of a file and to reduce disk seek times. However, while this additional processing generated by the file system stack may be acceptable for disk-based storage (CPU and main memory are thousands of times faster than disk), on NVM-based storage, 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 on itself (see Section IV-C1). 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 [13], [14], [136].

Some components of the storage subsystem are known to present significant processing overhead, most notably the device driver layer. Block drivers and schedulers traditionally implement strategies to optimize disk accesses, such as enqueuing and reordering I/O requests. During the design of Moneta [12], multiple sources of overhead in this level are identified including inefficient scheduling procedures, copies from and to user address space and the processing of interrupts. Moneta reports a significant per-operation latency reduction when completely bypassing the scheduling process. The issue with interrupts is also explored by Yang *et al.* [140]: when the underlying block device is fast enough, accessing it asynchronously is not always the best option due to the significant overhead of the block layer, especially when processing I/O related interrupts in the CPU (I/O request is enqueued and the CPU is interrupted when the operation is done). The study reports that in many cases (e.g., smaller or fine-grained I/O operations) simply performing I/O operations synchronously and polling the device awaiting the I/O completion is more efficient than performing I/O asynchronously.

Moneta-D [13] 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 entering kernel mode. Data is accessed through channels provided by a user-space 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.

As mentioned in Section II-B, some file system designs [120], [130] rely on user-space code to avoid the overhead caused by the I/O stack of the operating system. Building a file system at user level has some key advantages to that end, such as minimizing the interaction with the OS, bypassing unnecessary kernel space layers and allowing more fine-grained data management. In SIMFS [119], 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. 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 page tables (see Figure 5). Therefore, when an application maps 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

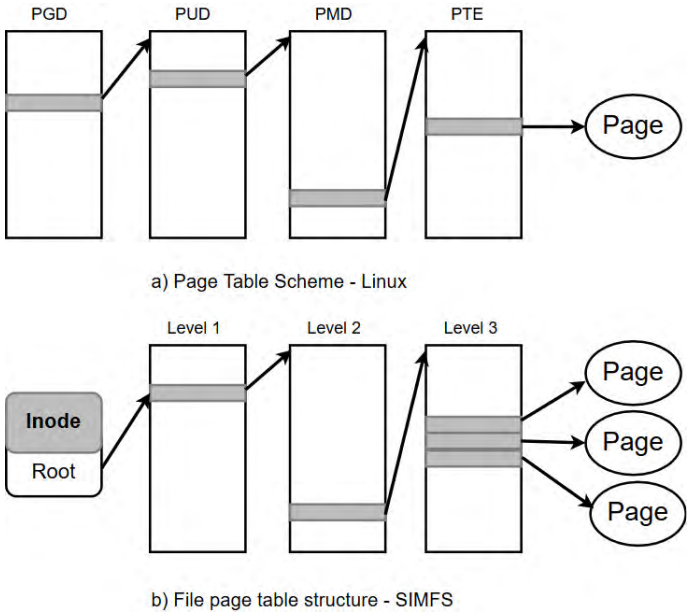


FIGURE 5. SIMFS index structure [119]. (a) Basic OS page hierarchical page table structure, (b) SIMFS file virtual address space index structure.

into the process address space, and the file's pages may be accessed directly with no further processing (for instance, no page faults).

6) RELIABILITY

Another highly desirable storage characteristic is reliability. Upcoming NVM technologies present some limitations in the endurance and reliability aspects. Reliability, as it is used in this paper, is related to loss of data caused by some physical failure in the device. For example, as memory chips density increase, the possibility of operations performed in the device cells creating noises in other cells becomes more likely [88]. Other problems, related to bus communication, may become a threat to reliability as well. For additional topics regarding integrity on NVM, see Sections V-B3 (on endurance) and V-A2 (on protection).

To improve the reliability of operations performed in both PM and SSD, error-correction codes are usually the solution [111]. These codes are used to verify the integrity of data after performing write operations, similarly 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 [75]. Thus even if data is corrupted due to hardware failures, a valid version of this persistent data may be recovered from a snapshot of an older version. In this case, solutions aim to find a balance between performance and reliability by, for example, adapting the frequency and granularity in which the checkpoint procedure is performed according to system parameters such as spatial locality and access patterns [114].

The Mojim system [142] improves reliability and high-availability by means of replication of NVM file system data across multiple nodes. It uses a two-tier architecture in which the primary tier contains a mirrored pair of nodes and the secondary tier contains one or more of secondary backup nodes with weakly consistent copies of data. Data is replicated across the network using an optimized RDMA-based protocol.

7) SCALABILITY

One of the main advantages of novel PM technologies over current volatile memories is their higher density. Combined with its low energy leakage, 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, to do it efficiently poses new challenges. For instance, providing address mechanisms for such a memory is not trivial. On the 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 [109]. 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.

Qiu and Reddy [109] describe a problem with SCMFS design, 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 2 MB superpages, reducing the number of addresses needed to designate large portions of data, optimizing the usage of TLB and its hit ratio. These superpages are used to address large files, while regular pages are used for the remaining data to

TABLE 8. List of common approaches for lifetime optimization and wear-leveling.

Name	Approach	Implemented Level
W-Buddy [118]	Remembering the number of writes per block	Memory Allocator
PM/NAND Flash Hybrid Architecture [124]	Out-of-place writes/log-based structure	Block Driver/FTL
Marching-Based Wear=Leveling [17]	Out-of-place writes/indirect pointers	File System
EqualChance [95]	Using logical address to rotate writes to physical addresses	Processor Cache Policies
Fine-Grained Wear-Leveling [110]	Using logical address to rotate writes to physical addresses	Memory Controller

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, however, it may compromise space efficiency on a small scale, cause write amplification, make memory protection more difficult and coarse-grained (larger pages means larger memory regions that must be placed under the same protection region) and adds overall complexity to the file system for the sake of translation performance.

The NOVA file system [138] addresses scalability by employing per-CPU metadata. Each CPU has its own journal and inode table to leverage the system's concurrency and to 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 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 of multiprocessors.

Kannan *et al.* [62] present pVM and discuss the drawbacks of adopting the Virtual File System (VFS, a transition/interface layer between the OS and the various file system implementations) as basis for future PM storage, such as scalability and flexibility limitations. According to their study, the main issues are that PM cannot be transparently allocated through the file system interface (e.g., when DRAM is exhausted) and that file systems do not distinguish memory allocated for persistent files and memory allocated to be used as operating memory, extending the volatile RAM. Furthermore, additional pages allocated from the VFS are managed by the file system, meaning that manipulating these pages involves frequent updates of metadata structures including superblocks, bitmaps and inodes that in turn has negative impact on processor cache and TLB. Given these limitations, pVM chooses to extend the kernel Virtual Memory subsystem (VM), allowing applications to allocate persistent memory through an API extension of the existing VM API (e.g., malloc and free functions). In this design, pages may be allocated across DRAM and PM seamlessly by the VM manager. For temporary (volatile) data, PM may be allocated transparently by the VM manager just like DRAM, without the VFS-related overheads. 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.

8) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Most issues discussed in this section have a strong connection with the costly write operations and erase operations characteristics of PM and SSD respectively. Since it is all based on a technology limitation, a common solution in terms of architecture is the hybrid memory approach, for instance, using DRAM for frequently updated data, thus reducing the impact of writes on NVM. In terms of system and software, the challenge is to identify hot and cold data design policies to ensure hot data is kept in more efficient (faster or more resilient) memory. Another concern that usually arises in hybrid memory is how to distinguish when PM is accessed as main memory (temporary data) and when PM is accessed as storage (long-term data). Distinguishing these access may benefit things like scheduling writes in the memory controller, reducing write latency for temporary data and more efficiently predicting memory access patterns. The challenges of software overhead and energy efficiency are also related. For instance, some works that aim to reduce the impact of software layers in traditionally storage systems, study the impact of reduced processor activity and memory footprint on power savings.

The most common method to work around NVM endurance limitation is out-of-place updates. Log-based file systems are a popular approach to the problem. These systems are also designed to provide other benefits such as low cost consistency and reliability. In general, out-of-place updates are an effective and simple solution, but may incur write amplification and garbage collection overhead. Therefore, there is an array of alternative solutions available in the literature, including hot/cold data filters and read-before-write, some of which may also benefit energy and performance (see Section V-C3). Table 8 enumerates most common approaches.

In disk-based storage, data integrity and fault tolerance are traditionally achieved through RAIDs and duplication. These approaches, although feasible in PM storage, pose a few new challenges since network and processor overhead may turn into a bottleneck [142]. Furthermore, issues such as endurance and byte-addressable interface must be taken in consideration as well, making traditional replication techniques incompatible with PM. Therefore, solutions in the area use mechanisms such as DAX, *mmap* and RDMA to explore byte-addressability and minimize overhead on replication solutions.

A common reliability issue is that of detecting and recovering from errors in memory level. In general, the OS handles errors and recovery in the memory level very differently from the faults in the storage level, treating faults like protection errors as critical [38]. While ECC and checkpointing may be helpful in some cases [5], additional mechanisms may be needed to treat memory faults on a per-case basis with more robustness.

The NVM scalability of a system may be limited by a series of factors including energy consumption, address translation limitations, cache inefficiency, concurrence limitation and memory security and protection. The analyzed scalability studies seek improvements on making address and protection more cache and TLB friendly, allowing systems allocated memory to grow transparently and partitioning of the entire physical address space. Besides these points, other more complex ideas are also described in studies such as decoupling protection from translation and adopting capability-based authorization [1]. At application level, object stores are usually the preferred solution for scalable storage instead of NVM file systems, for example, this is due to their flat name space and simpler structure and metadata [24] (see Section IV-A1).

C. NON-FUNCTIONAL - QUALITATIVE

This section explores topics on non-measurable requirements in NVM systems in general such as consistency and reliability.

1) CACHE CONSISTENCY

While many studies explore data and metadata consistency in terms of transactions and atomic updates (see Sections V-C2 and IV-A2) [37], [47], [144] some focus on a different type of consistency: integrating PM with the other layers of the memory hierarchy, most notably the processor cache. Since cache and buffers are typically volatile, during writes to PM data retained in cache lines may be lost upon system failures, leaving data in PM only partially updated. Additionally, writes may be reordered before reaching the PM layer, which in turn may compromise consistency mechanisms like journaling (e.g., if metadata is committed to PM before the actual data is persisted and the system crashes in the meantime). Hence, additional mechanisms are needed to ensure that cached data is correctly written back to PM.

The most common method for dealing with these problems is by issuing (explicitly or not) barrier and flush commands to the cache to ensure the persistence of cache lines and the order in which data will be written to PM. SCMFS [136], HEAPO [48] and PMFS [37] use a combination of *mfence* and *clflush* to ensure ordering and consistency. The *clflush* operation evicts the 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 increased traffic caused by periodic flushes plus the overhead of the ordering instructions may significantly degrade performance [144].

2) CONSISTENCY GUARANTEE

In order to avoid system corruption and data loss through multiple writes and erases, most storage solutions implement mechanisms to ensure consistency. This means that data stored in a physical medium, by a file system, database or object store, must be consistent at all times, 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 maintaining overall consistency, they are among the main sources of overhead in storage systems [14]. 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 at least two writes to the physical device (write amplification).

Consistency mechanism designs for PM file systems are among the most abundant topics addressed by the studies analyzed in this survey. Usually studies choose to adopt traditional mechanisms, like journaling and shadow paging tailoring them to obtain improved performance in PM. An example of fine-grain (e.g., 128 bytes per log entry) metadata journaling is presented by Chen *et al*. [20]. FSMAC creates multiple versions of the metadata before committing data to in-place update, enabling the operation to be undone. PMFS [37] uses a similar method of fine-grain journaling for metadata. A two-level logging scheme is proposed by Hwang *et al*. [49], mixing a fine-grain journaling mechanism with a log-based file system structure. In-memory Write-ahead Logging (IMWAL) [115] involves reusing the data log to update the database file by remapping memory addresses: it atomically updates metadata to point to newly updated pages instead of performing two full copies (once to the log and another to the original page). In cases where pages are only partially updated, partial page copies are made to merge new and old data in a single page. This approach, in turn, reduces the number of writes required for a commit operation when compared to traditional write-ahead logging methods. 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.

In a more generic approach, Kiln's [144] 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 are written to a PM cache in a transactional fashion. When the transaction is committed, the remaining volatile cache lines belonging to that transaction are flushed to the PM cache and the transaction is committed. When cache lines are evicted from the PM cache

they are copied back to the PM storage. This approach also needs additional hardware and modifications to the cache controller.

Non-volatile memory programming libraries such as Mnemosyne and Atlas [16], [24], [129] also provide their own built-in consistency methods. These methods usually take advantage of fine-grained data management and the high-level knowledge of the application to make consistency smarter and therefore can be more efficient than more traditional file system approaches. It is also common for these systems to use write-ahead logging combined with flushes to persistent memory to provide transactions with ACID properties at application level.

3) DATA PLACEMENT

Another similar and common trend in hybrid memories is to use fast memory (PM 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 fast and volatile memory or in slower, higher-capacity persistent memory [30], [112], [116], [141]. This strategy must identify critical data and determine whether they are temporary or must be persisted.

One example of block placement strategy is the Rank-aware Cooperative Caching (RaCC) block placement algorithm [112]. RaCC assumes an architecture featuring a PM/DRAM hybrid cache and works to optimize its utilization and performance. The 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 assumed to be faster than current PM technologies. When the number of accesses to an object cached in PM 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 by Dai *et al.* [30] 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 is polynomial in time and space. Conquest [131] is a file system design for a PM/disk hybrid storage. Conquest chooses a simple approach for data placement: it assumes that small files are responsible for most of the storage traffic. Using an arbitrary threshold (1 MB) to distinguish ''small'' from ''large'' files, the file system prioritizes storing larger files sequentially in the disk and small files in PM. It is relatively simple mechanism compared to other block placement algorithms, which also means less overhead and complexity on the storage management. The results show that Conquest may be up to 28 times faster than traditional file systems such as Ext2 and ReiserFS when performing random access.

The pVM system [62], adopts the persistent virtual memory model allowing page allocation and data placement across different memory technologies significantly easier. pVM treats PM as a NUMA node to both account for the difference in bandwidth between DRAM and PM and also to make the transition to the pVM model simpler. This design allows pVM to be extended to support more sophisticated data placement policies suitable to different PM technologies. While the system itself does not present data placement algorithms or complex heuristics for page placement, it implements a few flexible page allocation policy options: *nvmreverts* (allocate NVM only when lacking DRAM pages), *nvmpreferred* (favors allocation on PM over DRAM) and *nvmbind* (forces the usage of PM space only).

X-Mem [36] considers the use of persistent memory to expand the total memory footprint used by applications. It allows users to define tags for allocated memory ranges (through a *xmalloc* call) in order to distinguish between different types of data structures and their respective priorities. Each data structure/tag is managed by its own specific allocator and is mapped into the application's virtual address space. From there, X-Mem uses a profiler to automatically determine whether these structures should be stored in DRAM or PM. The profiler takes into consideration characteristics such as access patterns (e.g., strided, sequential, random) and frequency of access per tag/data structure. Figure 6 shows the basic X-Mem structure as it manages the storage of 3 different data types.

4) DATA DUPLICATION

In traditional systems, a single chunk of data may be replicated at multiple levels of the memory hierarchy simultaneously (disk, disk cache, main memory, processor cache, etc). Moving these copies around is necessary both for performance reasons and because different layers address data in different ways (blocks, pages and cache lines, for example). In an NVM-enabled architecture, performing memory to memory copy (e.g., for page and buffer cache), between PM and DRAM would be a source of unnecessary overhead, since data can be accessed directly from the PM device with DRAM-like speed. The studies listed under this category are concerned with reducing such redundancy in data, identifying where it is still needed and where it can be eliminated.

PMFS [37] integrates the execute-in-place (XIP) [128] 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 PM to DRAM. The same applies to mapped files: page descriptors are allocated and point to the mapped file in the PM storage, but no pages are actually copied to the page cache. Data is directly mapped in the process user space to be accessed and updated by the processor.

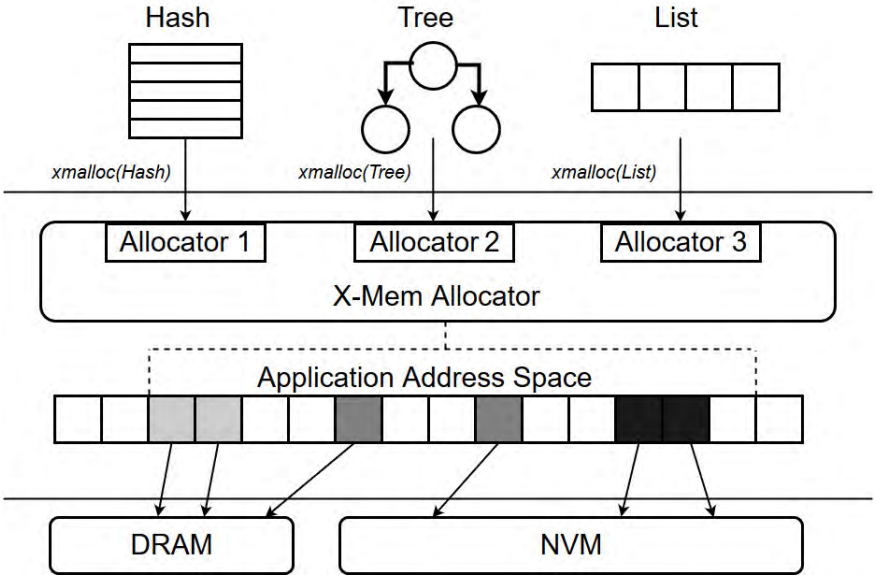


FIGURE 6. X-Mem structure [36]. Different types of data structured are handled differently by the OS and spread through DRAM and PM (NVM).

TABLE 9. Main system designs of NVM storage and their approaches to ensure consistency.

Name	Basic Consistency Approach	Cache Consistency Approach
Mnemosyne [129]	Journal/Logging/Shadow Paging	Barrier and Flush Commands/ Write Through Cache
PMFS [37]	Journal/Logging	Barrier and Flush Commands
BPFS [25]	Copy-on-Write/Shadow Paging	Epoch Barriers
NOVA [138]	Log Structured Files	Barrier and Flush Commands/ Write Through Cache
NV-Heaps [24]	Checkpoint/Full copy	None

5) CROSS-CUTTING COMPLEMENTARY INSIGHTS

Approaches for maintaining data consistency and integrity are numerous in the papers analyzed and their details may vary greatly. Table 9 presents how some of the main storage solutions in the literature ensure their data consistency. Consistency enforcement is one of the major differences between in-memory file systems (file systems that reside in volatile main memory, i.e., RAMFS) and NVM file systems. Notably, the issue goes beyond traditional storage consistency in PM file systems due to the cache consistency issue. Thus, consistency solutions seek to: (a) integrate the consistency mechanism with processor cache properly, (b) reduce processor overhead (compared to traditional mechanisms) and (c) minimize write amplification. Cache consistency in turn aims to avoid invalidating and flushing cache lines periodically, which may harm the system performance. It usually does this by means of relaxing ordering and dependency constraints or simply bypassing the cache when writing data back to memory. System-level solutions for the cache issue are very limited since cache controlling is mostly a hardware responsibility. Furthermore, more advanced solutions, such as epoch barriers may require modified hardware.

As we discussed throughout this paper, hybrid memory combining DRAM and PM (sometimes PM and SSD) is a popular approach to improve everything from energy consumption to device lifetime (see Section V-B). Regardless of the goal of the hybrid architecture, data placement policies are always a constant need. This is because simply using DRAM as a cache is suboptimal [36] and classic cache policies (LRU, LFU, etc.) cannot be used efficiently in this architecture due to the lack of information provided by hardware [80]. Overall the goals of the proposed data placement algorithms are: (a) to make the faster memory tier (DRAM or PM depending on the architecture) absorb as much write traffic as possible, (b) to distinguish temporary/main memory traffic from long-term/storage traffic and (c) to optimize the usage of limited amounts of fast memory.

VI. INDUSTRY TRENDS

This section is dedicated to recent topics, trends and solutions discussed in the computer industry regarding the adoption of existing and upcoming PM technologies. We start by understanding the industry needs that drive PM adoption. The amount of data stored and exchanged by today's applications has been growing at an unprecedented 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.

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 [108], 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.

A. TOOLS AND STANDARDS

The ACPI (Advanced Configuration and Power Interface) specification [2] 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 to provide information about features, configuration and addresses of PM devices. The NFIT describes PM regions, provides hints to make efficient use of cache flushes necessary to ensure durability of write operations to these regions, and the definition of block data window regions in case apertures are required to access the PM. The JEDEC Byte Addressable Energy Backed Interface standard [52] specifies the low-level access interface for PM devices. It is intended to simplify BIOS and PM 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 [145]. 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.

Another feature provided by PMEM is the block mode access to PM. As the name suggests, in block mode data is transferred to PM in blocks, in a similar fashion to block drivers. The block-granularity access, despite 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 performing direct access to PM. Accessing PM in block mode allows errors during access (e.g., ''bad blocks'') to be treated by the kernel, while managing errors when accessing PM 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 PM is the DAX (Direct Access) functionality [28]. The concept behind DAX is very similar to the concept of XIP (eXecute In Place), employed by some of the file systems discussed previously [37], [100]. The principle of DAX is to bypass Linux page cache, avoiding additional copies of data that would only represent unnecessary overhead, considering that the storage is built over PM. With DAX, PM 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 PM access mechanisms with modern and mature file systems designs.

B. ARCHITECTURE SUPPORT AND LIMITATIONS

Even though PM presents highly desirable attributes, such as low latency and high density, architectural support for these memories is still missing. A good example of this fact is the limited physical addressing capacity of current processors. An address space containing tens of terabytes (or even petabytes) of PM is well beyond what today's processors are capable of supporting [38]. Additional virtual and physical address bits need to be implemented for large-scale memory. Currently, the workarounds consist of mapping windows of PM regions into the physical address space of CPUs. More importantly, the overall implications of scaling memory to that amount are still unclear: increased occurrence of TLB and cache misses, increased overhead of memory zeroing and copying large amounts of memory. Ultimately, memory scaling is an open challenge that processors designs must cope with.

The processor caches are also not optimized for PM, some of which have already been discussed in Section VII-J. Existing barrier and cache flush instructions, while 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 in their new processors, *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. The *pcommit* instruction's goal was to ensure that writes accepted by the memory controller are committed to the persistent memory synchronously, however, it was deprecated since all upcoming memory controllers will already have hardware mechanisms to enforce this.

Another example of processor support for PM is the addition of memory protection keys (see Section V-A2). It adds a 32 register to the processor that is used to define 16 sets of 2-bit page access permissions (a read and a write bit). Each set is mapped to a key that may be assigned to a group of pages through their entries in the page table. MPK allows for a more efficient implementation of *mprotect*, that could be used to lock PM ranges and protect it from stray

writes while avoiding TLB flushes and not compromising performance. The TLB is yet another point of improvement in current architecture. The address translation buffer is critical to address translation (and, therefore, to the whole system's performance) and it is a scalability bottleneck on systems with huge amounts of memory. Despite not being a new topic, already a common issue in virtual memory management as well, the emergence of huge memory systems and NVM file systems put this challenge in a new perspective.

C. PROGRAMMING MODELS

As briefly discussed in Section VII-D, traditional access methods and programming models may not be the best fit for PM storage due to their unique characteristics, which led to the proposal of new programming models and access methods aiming to explore NVM features [96], [129]. The Storage and Networking Industry Association (SNIA) has defined and published the NVM Programming Model (NPM) specification to provide some directions for developers to provide common and extensible PM access model. The specification is also useful for users to understand what can be expected and what operations can be performed over PM systems. The NPM 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, NPM also provides a few use cases to illustrate the usage of the specified operations and describes a few directions to make programming interfaces compliant with the specification.

The NVM Library (NVML) [96] is a set of open libraries designed to provide applications with easy and efficient access to PM. The NVML follows the design principles that are specified in the NPM, 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 low-level functions, like cache flushing, and optimized file mapping. In higher-level libraries, NVML supports atomic transactions, persistent pointers and lists, as well as synchronization for multithreading. Finally, NVML also provides a C++ version of the API, allowing more intuitive and robust object-oriented programming over PM.

VII. FUTURE DIRECTIONS

This section is dedicated to analyzing the main trends that are the most likely to be targeted for future advances in the area of NVM file systems and to provide insights in their relevance and their direction. We build this analysis based on the amount of effort dedicated to each topic as well as their content and our vision and opinion regarding the impact of their proposals. 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.

The conclusions presented here are based on 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 paper, the focus of this discussion is on the storage level and its responsibilities. Also, we target architectures where PM is located at the same level of today's DRAM (memory bus). Thus, some contributions like user level programming models, SSD focused solutions and alternative architectures may not be taken into account in this discussion.

Table 10 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. For this illustration we take the following systems:

- *TMPFS*: a well-known in-memory file system usually employed to store temporary files on (volatile) main memory. It uses OS internal VFS structures to allow user-level code to manipulate in-memory data using file system semantics [121].
- *DAX*: short for Direct Access. It is a mechanism implemented in the Linux OS to allow traditional file systems to bypass storage layers (for instance, the page cache) when working with NVM [28].
- *PMFS*: a file system designed by Intel to provide efficient access to NVM storage using traditional file system API [37].
- *BPFS*: an NVM file system that implements NVM optimized consistency techniques such as short-circuit shadow paging and epoch barriers [25].
- *Mnemosyne*: a user-level library that provides efficient and reliable access to persistent memory through a more memory-like interface, offering features such as persistent regions and variables [129].
- *Muninn*: an object-based key-value store designed to be a light and flexible solution to exploit NVM characteristics while hiding its intricacies [61].
- *HEAPO*: an object-based persistent heap implementation designed for NVM with features such as undo-logging and native memory protection [48].
- *SCMFS*: a lightweight NVM file system built over a contiguous address space by extending the OS virtual memory subsystem [136].

The last column of the table presents a simple conclusion about the maturity of each specific topic. It helps to illustrate the current level of development of each one of the topics we have identified in this study, indicating which topics need further exploration and which are closer to (or already) presenting deployable, working solutions. Besides the contributions listed in the table, we also take in account the

TABLE 10. Comparison of studied storage solutions.

Problem	NVM System Samples								Maturity Level
	TMPFS	DAX	PMFS	BPFS	Mnemosyne	Muninn	HEAPO	SCMFS	
Consistency Guarantee	NO	YES	YES	YES	YES	YES	YES	YES	HIGH
Atomicity	NO	YES	YES	YES	YES	NO	YES	YES	HIGH
Endurance	NO	NO	NO	NO	NO	YES	NO	NO	HIGH
Access Interface	NO	NO	NO	NO	YES	YES	YES	NO	MEDIUM
Asymmetric Latency	NO	NO	NO	NO	NO	NO	NO	NO	MEDIUM
Metadata Management	NO	NO	YES	YES	YES	YES	YES	YES	LOW
Software Overhead	NO	YES	YES	YES	YES	NO	YES	YES	HIGH
Block/Page Allocation	NO	YES	YES	YES	YES	YES	YES	YES	MEDIUM
Memory Protection	NO	N/A	YES	YES	N/A	NO	YES	YES	LOW
Cache Consistency	N/A	NO	YES	YES	YES	NO	NO	YES	LOW
Cache Optimization	NO	NO	NO	YES	NO	NO	NO	NO	MEDIUM
Reliability	YES	YES	YES	NO	NO	NO	NO	NO	HIGH
Write Amplification	NO	NO	NO	YES	NO	YES	NO	NO	MEDIUM
Data Placement	NO	NO	NO	NO	NO	NO	NO	NO	LOW
Access Transparency	YES	YES	YES	YES	NO	YES	NO	YES	HIGH
Data Duplication	YES	YES	YES	NO	YES	NO	YES	YES	HIGH
Scalability	NO	YES	NO	NO	NO	NO	NO	YES	LOW

amount and quality (measured by citations) of the research papers analyzed in this survey (see Figure 1) and the advances in industry (see Section VI) to determine each topic's development level. We separate topics in three maturity levels:

- *High*: issues that are at a relatively mature level of development, having relevant papers published in the area and satisfactory solutions already proposed and successfully 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.
- *Medium*: topics that have been extensively studied but have no definitive solution in sight. Usually, these are complex or poorly understood issues, that demand either changes in the architecture or high effort and complex solutions.
- *Low*: topics that are known to be relevant but have not received as much attention as the others or have not been extensively explored in the literature. These topics have the most research potential and are the most likely to drive innovative studies at the moment.

The remaining topics we consider to be non-critical, (i.e., space efficiency, energy efficiency, garbage collection, fragmentation, persistent cache, parallelism, and mounting time). These issues are either expected to be solved when the technology reaches a certain level of maturity, or they simply do not represent a threat to an architecture where CPU access NVM directly. These issues may eventually become points of interest, as NVM-based systems evolve and change, but for the foreseeable future, they do not seem to be critical issues.

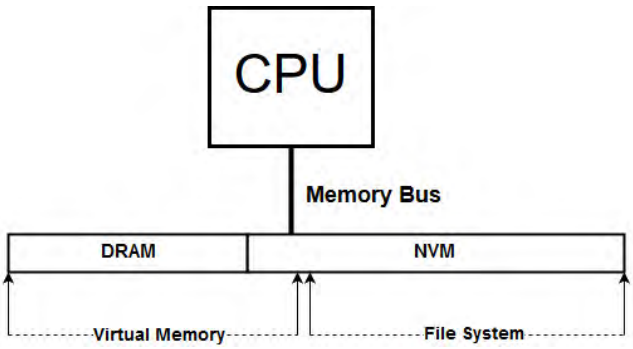


FIGURE 7. Target architecture - single address space model.

In spite of the studies discussed in this paper being related to different architectures and having 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 7 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 presents multiple challenges such as cache reordering and persistence of stray writes.

A. CONSISTENCY GUARANTEE

It is clear that there is still plenty of space for new methods of improving file system consistency performance and reliability. Nevertheless, there is already a large array of solutions available in the literature. In fact, many of these solutions are improvements over classic consistency methods, such as fine-grained journaling, log-structured file systems and short-circuit shadow paging. Furthermore, several studies hint that methods of data consistency will evolve as the file systems and metadata themselves evolve and this evolution depends on other factors (such as metadata, addressing and scalability), hence it might resurface as a hot topic in the future. However, it seems that existing solutions cited previously already fill a major part of the requisites of an efficient consistency mechanism and may be adopted in upcoming NVM storages.

B. ATOMICITY

Atomicity is yet another topic that is reminiscent of traditional file systems and was again extensively explored on SSD recently. The concepts of atomic operations and transactions have been studied for decades and are relevant to most storage models, including file systems and databases. Like the methods to ensure consistency, methods of providing atomic operations on NVM file systems are to a large extent based on traditional mechanisms.

The issue of atomicity is closely related to that of file system and (most precisely) user data consistency and, hence, their developments are highly linked to each other. Furthermore, the success of atomicity techniques depends on other (arguably) more pressing matters such as ordered cache flushes. Finally, current solutions, such as atomic mappings and lightweight transactions may still have a long way to go, but they already serve their purposes and show how atomicity in an NVM environment should be treated.

C. ENDURANCE

Much research has already been dedicated to this matter (see Figure 1) since the impact of limited endurance of NVM is a critical factor for its adoption. Furthermore, as mentioned in Section II-A, some types of NVM already present a satisfactory endurance (for storage level usage) and the lifetime of the remaining technologies is 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). Therefore, we argue that it is relatively safe to assume that, in the long term, dealing with endurance will not be as challenging as it was in the recent past and file systems will be able to focus on the aspects that really matter to users, such as performance, security and reliability.

D. ACCESS INTERFACE

Even though many PM file systems [25], [37] 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 PM, but no definitive framework or model exists 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 PM-enabled environments, the storage interface and access methods must evolve as well.

Re-evaluating the way storage is accessed is one the most critical and interesting questions related to PM-enabled systems, and it has potential for innovation and reinvention such as on file system semantics. This particular aspect of PM storage is closely attached to the evolution of modern application development and development frameworks that indicates that this is a good time to review the role of file systems in current and future applications. Furthermore, the adoption of PM brings storage much closer to the main memory, opening new possibilities for data management and enabling mechanisms and models previously unfeasible. These are indicators that the interface of applications to NVM will certainly be one of the most relevant NVM-related topics and it will not be restricted to the file system level.

E. ASYMMETRIC LATENCY

From an operating system perspective, currently there are not many options available to reduce the impact of slow write/erase operations on the system's performance. Usually the solution for storage is found in minimizing write amplification, and employing faster memories for buffering writes to files. Assuming that the latency of upcoming NVM technologies will not be improved naturally at hardware level, optimizing the write performance of NVM should be considered a critical point as we aim for NVM-based storage to get as close as possible to today's volatile memory layers. The number of factors that impact write performance are numerous and their relationship must be carefully considered on future studies as well.

F. METADATA MANAGEMENT

Upcoming PM introduces a lot of questions, like whether hierarchical or flat name space would be a best fit and whether files are even the best abstraction to work with on memory-based storage. Despite the fair amount of research that has been invested in metadata designs, most of them are,

at some level, based on structures developed for disks and SSD, and many of the aforementioned questions remain with no definitive answer.

Traditional metadata structures are designed for block-based file systems, which may be appropriate for SSDs but may become an issue in PM-based storage [62]. It impacts almost every aspect of the file system design and raises questions such as whether existing file abstractions are even adequate for NVM [5]. Along with the improved interfaces for future NVM-based storage, managing metadata is one of the most flexible topics in this list and has a huge space for exploration and experimentation. Hence, trying to predict what NVM storage metadata will look like in the future is speculative at this point. We do, however, agree with observation NVM should employ smaller metadata structures with focus on fine-grained manipulation and interfere with the application-data interaction as little as possible. It is also likely that the concepts in storage metadata will get mixed with OS and applications memory structures to bring persistent data closer to applications and to provide more flexible and efficient methods to manipulate data.

G. SOFTWARE OVERHEAD

Much of the software overhead in traditional file system models has been already eliminated in NVM file systems by simply avoiding layers like I/O schedulers and page cache. Further performance improvement is possible by also bypassing caching procedures and mapping pages directly to user space, using existing mechanisms such as XIP [37], [67]. Additionally, more fine-tuned improvements may be implemented by avoiding low-level procedures, like entering kernel mode, avoiding switching permission levels using memory mappings and performing synchronous operations rather than using interrupts.

With major sources of overhead out of the way, we believe that identifying and eliminating points of overhead from this point onwards will be trickier as the critical path of accessing PM moves to a more low-level and hard to optimize code (e.g., page table walk). The file system overhead, is expected to become less prohibitive for PM as expensive mechanisms, such as consistency, block allocation and wear-leveling algorithms evolve and become more sophisticated. On the kernel side, the future of memory-related mechanisms, specially the address translation process, that takes a major role in the process of accessing data, is uncertain. Much like metadata, it seems that in the most natural transition for PM storage addressing from today's model to the architecture presented in Section 7, file address space may get mixed with OS and process memory addressing (involving page tables). However, we argue that the addressing issue has a bigger impact on other aspects of NVM, such as scalability, that are currently more critical than OS overhead.

H. BLOCK ALLOCATION

Block allocation algorithms and policies for NVM storage are numerous and well established, but there are still unexplored issues regarding this process and others that may arise in the future. As we have seen in this study, most NVM block allocation algorithms and policies are designed to mitigate fragmentation and improve lifetime. While dealing with these matters in the allocation process may be redundant (if not simply unnecessary), block allocation is one of the core concepts of a file system. Optimizing it is essential when dealing with most storage-related challenges, such as file system concurrency and address translation. Existing solutions for these problems are usually complex and their impact on the overall system is still not completely clear. As the requirements of NVM file systems become clearer over time, the responsibilities of allocators might evolve and more robust solutions will be needed. Even though these requirements are still being investigated, solutions in this area will need to be lightweight, highly concurrent and very flexible.

I. MEMORY PROTECTION

Memory protection is not a new challenge in the OS area, however, it gains a completely new outlook when taking persistence in consideration. Currently, the trend in protecting PM exposed directly to the CPU seems to be switching the write permission of memory pages on and off by exploring the protection bits of page table entries and processor registers [27]. Others more recent approaches revisit capability-based systems to provide efficient fine-grained protection [1]. However, these methods do not seem organic solutions for the problem, and while some of them may incur significant overhead, others depend on hardware innovations and experimental methods. Clearly, memory protection should be seen as a critical issue and that a robust definitive solution might involve rethinking OS and file system level addressing structure and how they interact.

J. CACHE CONSISTENCY

In an ideal scenario, for the architecture in Figure 7, processor caches would be persistent to avoid all the issues and drawbacks of having a volatile cache interacting with an NVM storage. However, due to current NVM limitations, like endurance and latency, a persistent processor cache does not seem to be feasible. While it would seem that using processor operations to orderly flush data from volatile caches to NVM and optimizing cache and processor to mitigate the impact of doing such would be a reasonable solution, the results show otherwise. One particular study that presents a comparison of different cache modes [7] shows that in some cases, using a combination of memory fence and flush along with write-back cache mode performs twice as worse as write-through and, in some scenarios, it may be worse than not caching data at all. The fact is that this line of solution does not seem to be very efficient and processor support for it is still quite lacking. These impacts must be further explored to fully understand the requirements and the impacts of the relationship between volatile cache and NVM storage. This knowledge is fundamental to properly design future cache management algorithms and policies. Additionally, new and

alternative mechanisms for more robust and transparent cache management (e.g., the epoch barriers [25]) constitute promising research in the NVM area.

K. CACHE OPTIMIZATION

Cache and buffer optimizations are usually designed to improve SSD storage. Hence, most of these optimizations do not apply to PM 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 PM efficiently (see Section VII-E) represent the main contribution in terms of optimizing cache usage. The other potentially significant contributor to improvement is the TLB. As we have already explored in this discussion, the address translation and memory protection are among the most complex and currently relevant topics that still deserve more thorough study. The TLB mechanism takes a major role in memory translation and therefore deserves special attention. Similarly to processor caches, the impact of novel PM solutions over the TLB are not sufficiently clear, and must be carefully considered in the design of future PM file systems.

L. RELIABILITY

Most reliability studies are concerned with low-level fault-tolerance such as error detection and correction codes and hardware primitives since corruption of data in either PM or SSD could be disastrous. Studies that do explore file system level reliability usually do it in terms of checkpointing algorithms and RAID-like solutions, which are both already well known concepts in today's industry. While it is clear that there is space for more NVM friendly fault-tolerance mechanisms, it does not seem like this is a particularly demanding field: studies are more concerned with lightweight consistency mechanisms and enforcing metadata consistency. This topic is expected to attract more research in the future (especially since data redundancy in NVM is still relatively unexplored to date), perhaps as alternatives for more complex consistency techniques. At its current state, reliability does not belong among the most critical issues.

M. WRITE AMPLIFICATION

With the transition to a self-contained and simpler model, NVM file systems already eliminate most of the write amplification presented in traditional file systems by, for instance, eliminating duplications and optimizing journaling and garbage collection. Furthermore, as applications are given more freedom to access persistent data directly and interaction with OS and file systems is reduced, write amplification will be reduced naturally. Techniques like fine-grained logs, direct mappings and atomic in-place updates contribute to reducing overall write amplification and may be used as model for more sophisticated methods in the future. While minimizing write amplification is important in terms of performance and energy consumption, it will become less of an issue in the future as file system designs become more flexible.

N. DATA PLACEMENT

In the target architecture, adopting a block placement policy may be useful to work efficiently with multiple memory layers or hybrid memory (DRAM combined with PM). While copying data from memory to 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. While some studies explore block placement techniques, only a few are actually designed for a hybrid memory layer similar to our target architecture. Understanding of the impacts, requirements and behavior of hybrid memory systems is still lacking. Furthermore, potential solutions such as identifying hot and cold data, are highly dependent on the type of application and their interaction with NVM, which makes generalizing it a non-trivial job. With that in mind, exploring the impacts of hybrid architectures and block placement in file systems have great potential for exploration and consider it to be a very interesting research topic.

O. ACCESS TRANSPARENCY

In the context of this work, transparency is related to compatibility and legacy systems accessing NVM storage in an efficient way with traditional interfaces. It should not be one of file system's concerns to ensure compatibility with technologies, but it is important for applications to rely on NVM file systems for a consistent behavior. However, exploring new ways of exposing data and bringing NVM closer to applications is currently a more pressing matter. Furthermore, as NVM devices enter the market, storage solutions to boost existing application's performance (with little or no modification required) will become more popular, and transparency will be an important requirement for these tools.

P. DATA DUPLICATION

In an architecture where the file system is merged with main memory, the amount of data duplication will be naturally 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 for copies or page swaps. Furthermore, mechanisms such as XIP that allow pages to be directly mapped into user space represent a solution that involves little to no data duplication at all. Therefore, state-of-the-art PM file systems have already reached a good level of deduplication including zero copy access to PM pages. The exception being cases where data must be replicated for reliability and consistency reasons, such as for out-of-place writes, for instance. In these cases, the balance between the memory footprint and reliability trade-off must be considered for each scenario.

Q. SCALABILITY

The impact of large amounts of PM on the memory bus and overall architecture is very little addressed and understood. When PM is attached to the memory bus, it will also use memory management resources, such as the memory controller, TLB, and other translation mechanisms. Additionally, the VFS design can be inherently cache and TLB inefficient, which makes in-memory file systems challenging to properly scale [62]. Since PM availability is still limited and emulation of PM-enabled environment is limited, it is unclear how these mechanisms will perform with large pools (petabytes) of memory.

Scalability is currently a very relevant topic and one of the most lacking in terms of research. This is probably due the fact that huge memory architectures are still not a concrete possibility and simulated models are complex and not reliable enough. Understanding the requirements for such architectures is essential since higher density and larger memory arrays are expected to be one of the biggest contributions of PM technology for computing. Scalability will certainly attract more attention in the future, even though robust scalable solutions might be more difficult to design and implement and might add significant complexity to the storage.

VIII. RELATED WORK

A couple of related studies [94], [133] may further complement the results shown in this paper. We have identified that these studies share some of our motivations and objectives, although they present different approaches and obtain different results. Those results may also reinforce some of the beliefs we present in this paper, including the relevance of topics such as wear-leveling, consistency costs and energy efficiency.

The survey presented in [94] provides an overview of NVM technologies, highlighting their limitations, characteristics and even presenting an extensive comparison between the many available options for storage memory technology. It also explores the implications and challenges of different hybrid memory approaches such as combining Phase Change Memory (PCM) with DRAM or NAND Flash. What is possibly the main contribution of the paper, however, is the categorization of the reviewed studies according to their objectives and/or used approaches, identifying challenges and their proposed solutions, similar to the work presented in this paper (see Section III-C) only with different parameters and points of view.

Another study [133] explores the challenges of adopting Phase-Change RAM (PCM) as storage, main memory and/or processor cache, summarizing existing solutions and classifying them. The survey first presents the main challenges of adopting NVM in any level of memory hierarchy and proceeds to discuss the many approaches present in the literature detailing the most relevant ones. The presentation of both challenges and approaches are structured according to their target level on the memory hierarchy (cache, main memory or long-term storage), similar to the summary depicted in Figure 4 but in more details. It also provides a comparison between the many techniques and models presented by the surveyed studies according to their area of impact or topic. Furthermore, although it presents relevant discussion on file systems and storage in general, the study is not focused on these particular approaches, choosing to contemplate a broader area.

In summary, the work presented in this paper basically differs from the previous papers by focusing on file system level topics and design (although many studies that go beyond file systems are also discussed) and by trying to give directions for researchers to identify topics that are currently hot or lacking exploration. This paper also adopts a systematic mapping structure, which may be useful for future research and provides some degree of transparency on the survey process.

IX. CONCLUSION

NVM technologies are a big promise in terms of high-performance scalable storage: they not only close the gap between block devices (HDDs and SSDs) and RAM but may potentially replace both in favor of a single-level memory hierarchy. They are also very flexible, as they can be used as main memory, block-based I/O storage or even as an in-memory storage. Thus, NVM is expected to be an effective solution for many of today's application challenges in data handling by bringing data closer to the processing unit.

That being said, the understanding relative to the impacts of a fast byte-addressable RAM storage on the overall architecture is still quite lacking. Operating system and hardware support for such storage models are still in a very immature state, even though many studies supporting NVM adoption have been conducted in both academia and industry. This is due to a number of factors: NVM availability is currently limited both on the market and as prototypes; pressure of large devices on memory bus is unprecedented; operating systems are optimized for slow block-oriented storage; and eliminating storage layers (such as block drivers and I/O schedulers) as well as persistence in RAM level present a series of security and reliability risks.

Furthermore, due to NVM's byte-addressability and low latency, the line between file systems and memory management gets blurred. Thus, as persistent data (in whatever form, e.g., files, objects, address ranges etc.) is integrated into the OS, the concepts of structures that are traditionally volatile and structures that are traditionally persistent usually get confused. In addition, there seems to be a conflict between the ideas of modeling data in NVM as memory structures versus modeling it as traditional files. Using a memory-driven approach avoids serialization of data from a programming and memory-friendly model to a more traditional file or database format. However, in some cases it may mean sacrificing things like portability, shareability and hierarchical structures, which are natural characteristics of file systems.

On the other hand, byte-addressable NVM enables a number of innovations such as persistent heaps, direct file

mappings and smart metadata indexation. Despite these solutions being in their early phase, it is possible to identify a few interesting trends, such as mapping NVM regions directly (XIP or DAX concept) or using user-level file systems to reduce the overhead of entering the kernel.

As we mentioned previously in this work (Section I), migrating current concepts and mechanics to an NVM-enabled architecture is going to be an iterative three-step process. While the NVM chips themselves are still not widely available in the market and are mostly accessed through adapted legacy interfaces (first step), there is already plenty of research and development projects in industry and academia that would fit the second and even the third steps described in Section I. As we have demonstrated in our results, much effort has been dedicated to the design of NVM-improved storage systems such as file system and object-based store libraries, and there is much interest on mechanisms such as mapping persistent memory in process space and orthogonal persistence.

Although it may seem that NVM storage is a distant reality, 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.

REFERENCES

[1] R. Achermann *et al.*, "Separating translation from protection in address spaces with dynamic remapping," in *Proc. ACM Workshop Hot Topics Oper. Syst.*, 2017, pp. 118–124.
[2] (2015). *Advanced Configuration and Power Interface Specification*. [Online]. Available: http://uefi.org/specifications
[3] A. Ames *et al.*, "Richer file system metadata using links and attributes," in *Proc. 22nd IEEE/13th NASA Goddard Conf. Mass Storage Syst. Technol.*, Apr. 2005, pp. 49–60.
[4] S. Baek, D. Son, D. Kang, J. Choi, and S. Cho, "Design space exploration of an NVM-based memory hierarchy," in *Proc. IEEE 32nd Int. Conf. Comput. Design*, Oct. 2014, pp. 224–229.
[5] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy, "Operating system implications of fast, cheap, non-volatile memory," in *Proc. 13th USENIX Conf. Hot Topics Oper. Syst.*, 2011, p. 2.
[6] O. Barbosa and C. Alves, "A systematic mapping study on software ecosystems," in *Proc. Workshop Softw. Ecosyst.*, 2011, pp. 15–26.
[7] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Implications of CPU caching on byte-addressable non—Volatile memory programming," HP Lab., Palo Alto, CA, USA, Tech. Rep. HPL-2012-236, 2012.
[8] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, "Makalu: Fast recoverable allocation of non-volatile memory," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2016, pp. 677–694.
[9] P. Biswas and D. Towsley, "Performance analysis of distributed file systems with non-volatile caches," in *Proc. 2nd Int. Symp. High Perform. Distrib. Comput.*, 1993, pp. 252–262.
[10] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy, "Overview of candidate device technologies for storage-class memory," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 449–464, 2008.
[11] A. M. Caulfield and S. Swanson, "QuickSAN: A storage area network for fast, distributed, solid state disks," in *Proc. 40th Annu. Int. Symp. Comput. Archit.*, 2013, pp. 464–474.
[12] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson, "Moneta: A high-performance storage array architecture for next-generation, non-volatile memories," in *Proc. 43rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2010, pp. 385–395.
[13] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," *ACM SIGARCH Comput. Archit. News*, vol. 40, no. 1, pp. 387–400, Mar. 2012.
[14] A. M. Caulfield *et al.*, "Understanding the impact of emerging non-volatile memories on high-performance, IO-intensive computing," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1–11.
[15] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program., Syst., Lang., Appl.*, 2014, pp. 433–452.
[16] D. R. Chakrabarti, H.-J. Boehm, and K. Bhandari, "Atlas: Leveraging locks for non-volatile memory consistency," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 433–452, 2014.
[17] H.-S. Chang, Y.-H. Chang, P.-C. Hsiu, T.-W. Kuo, and H.-P. Li, "Marching-based wear-leveling for PCM-based storage systems," *ACM Trans. Design Autom. Electron. Syst.*, vol. 20, no. 2, p. 25, 2015.
[18] F. Chen, D. A. Koufaty, and X. Zhang, "Understanding intrinsic characteristics and system implications of flash memory based solid state drives," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 37, no. 1, pp. 181–192, 2009.
[19] F. Chen, M. P. Mesnier, and S. Hahn, "A protected block device for persistent memory," in *Proc. 30th Symp. Mass Storage Syst. Technol.*, 2014, pp. 1–12.
[20] J. Chen, Q. Wei, C. Chen, and L. Wu, "FSMAC: A file system metadata accelerator with non-volatile memory," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, May 2013, pp. 1–11.
[21] K. Chen, R. Bunt, and D. Eager, "Write caching in distributed file systems," in *Proc. 15th Int. Conf. Distrib. Comput. Syst.*, 1995, pp. 457–466.
[22] R. Chen, Y. Wang, J. Hu, D. Liu, Z. Shao, and Y. Guan, "Virtual-machine metadata optimization for I/O traffic reduction in mobile virtualization," in *Proc. Non-Volatile Memory Syst. Appl. Symp.*, 2014, pp. 1–2.
[23] Z. Chen, Y. Lu, N. Xiao, and F. Liu, "A hybrid memory built by SSD and DRAM to support in-memory big data analytics," *Knowl. Inf. Syst.*, vol. 41, no. 2, pp. 335–354, 2014.
[24] J. Coburn *et al.*, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *SIGARCH Comput. Archit. News*, vol. 39, no. 1, pp. 105–118, 2011.
[25] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proc. 22nd ACM Symp. Oper. Syst. Princ.*, 2009, pp. 133–146.
[26] J. Corbet. (2014). *LFCS: Preparing Linux for Nonvolatile Memory Devices*. [Online]. Available: http://lwn.net/Articles/547903/
[27] J. Corbet. (2015). *Memory Protection Keys*. [Online]. Available: http://lwn.net/Articles/643797/
[28] J. Corbet. (2014). *Supporting Filesystems in Persistent Memory*. [Online]. Available: http://lwn.net/Articles/610174/
[29] H. Dai, M. Neufeld, and R. Han, "ELF: An efficient log-structured flash file system for micro sensor nodes," in *Proc. 2nd Int. Conf. Embedded Netw. Sensor Syst.*, 2004, pp. 176–187.
[30] P. Dai, Q. Zhuge, X. Chen, W. Jiang, and E. H.-M. Sha, "Effective file data-block placement for different types of page cache on hybrid main memory architectures," *Des. Autom. Embedded Syst.*, vol. 17, nos. 3–4, pp. 485–506, 2014.
[31] D. Das, D. Arteaga, N. Talagala, T. Mathiasen, and J. Lindström, "NVM compression—Hybrid flash-aware application level compression," in *Proc. 2nd Workshop Interact. NVM/Flash Oper. Syst. Workloads*, 2014, pp. 1–10.
[32] I. H. Doh, J. Choi, D. Lee, and S. H. Noh, "An empirical study of deploying storage class memory into the I/O path of portable systems," *Comput. J.*, vol. 54, no. 8, pp. 1267–1281, Aug. 2011.
[33] I. H. Doh, J. Choi, D. Lee, and S. H. Noh, "Exploiting non-volatile RAM to enhance flash file system performance," in *Proc. 7th ACM/IEEE Int. Conf. Embedded Softw.*, Sep. 2007, pp. 164–173.
[34] I. H. Doh *et al.*, "Impact of NVRAM write cache for file system metadata on I/O performance in embedded systems," in *Proc. ACM Symp. Appl. Comput.*, 2009, pp. 1658–1663.
[35] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang, "Rethinking benchmarking for NVM-based file systems," in *Proc. 7th ACM SIGOPS Asia–Pacific Workshop Syst.*, 2016, pp. 20:1–20:7.
[36] S. R. Dulloor *et al.*, "Data tiering in heterogeneous memory systems," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, pp. 15:1–15:16.
[37] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–15.

[38] P. Faraboschi, K. Keeton, T. Marsland, and D. Milojicic, "Beyond processor-centric operating systems," in *Proc. 15th Workshop Hot Topics Oper. Syst.*, 2015, pp. 1–7.

[39] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 439–447, 2008.

[40] M. M. Fu and P. Dasgupta, "A concurrent programming environment for memory-mapped persistent object systems," in *Proc. 17th Int. Comput. Softw. Appl. Conf.*, 1993, pp. 291–298.

[41] E. Giles, K. Doshi, and P. Varman, "Bridging the programming gap between persistent and volatile memory using WrAP," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2013, pp. 1–10.

[42] E. R. Giles, K. Doshi, and P. Varman, "SoftWrAP: A lightweight framework for transactional support of storage class memory," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, May/Jun. 2015, pp. 1–14.

[43] K. M. Greenan and E. L. Miller, "Reliability mechanisms for file systems using non-volatile memory as a metadata store," in *Proc. 6th ACM/IEEE Int. Conf. Embedded Softw.*, Oct. 2006, pp. 178–187.

[44] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei, "Software persistent memory," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, p. 29.

[45] J.-I. Han, Y.-M. Kim, and J. Lee, "Achieving energy-efficiency with a next generation NVRAM-based SSD," in *Proc. Int. Conf. Inf. Commun. Technol. Converg. (ICTC)*, 2014, pp. 563–568.

[46] Y. Hu, T. Nightingale, and Q. Yang, "RAPID-cache reliable and inexpensive write cache for high performance storage systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 13, no. 3, pp. 290–307, Mar. 2002.

[47] J. Huang, K. Schwan, and M. Qureshi, "NVRAM-aware logging in transaction systems," *Proc. VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[48] T. Hwang, J. Jung, and Y. Won, "HEAPO: Heap-based persistent object store," *ACM Trans. Storage*, vol. 11, no. 1, pp. 3:1–3:21, 2015.

[49] Y. Hwang, H. Gwak, and D. Shin, "Two-level logging with non-volatile byte-addressable memory in log-structured file systems," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, p. 38.

[50] S. Im and D. Shin, "Differentiated space allocation for wear leveling on phase-change memory-based storage device," *IEEE Trans. Consum. Electron.*, vol. 60, no. 1, pp. 45–51, Feb. 2014.

[51] N. S. Islam, M. Wasi-Ur Rahman, X. Lu, and D. K. Panda, "High performance design for HDFs with byte-addressability of NVM and RDMA," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 8:1–8:14.

[52] (2015). *JEDEC Byte-Addressable Energy Backed Interface*. [Online]. Available: http://www.jedec.org/standards-documents/docs/jesd245

[53] N. Jeremic, G. Mühl, A. Busse, and J. Richling, "Operating system support for dynamic over-provisioning of solid state drives," in *Proc. ACM Symp. Appl. Comput.*, 2012, pp. 1753–1758.

[54] R. Jin, H.-J. Cho, S.-W. Lee, and T.-S. Chung, "Lazy-split B^+-tree: A novel B^+-tree index scheme for flash-based database systems," *Design Autom. Embedded Syst.*, vol. 17, no. 1, pp. 167–191, 2013.

[55] T. Johnson and T. A. Davis, "Parallel buddy memory management," *Parallel Process. Lett.*, vol. 2, no. 4, pp. 391–398, 1992.

[56] A. Joshi, V. Nagarajan, M. Cintra, and S. Viglas, "Efficient persist barriers for multicores," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 660–671.

[57] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "FRASH: Exploiting storage class memory in hybrid file system for hierarchical storage," *ACM Trans. Storage*, vol. 6, no. 1, pp. 3:1–3:25, 2010.

[58] M. Jung *et al.*, "Exploring the future of out-of-core computing with compute-local non-volatile memory," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2014, pp. 125–139.

[59] D. Kang, S. Baek, J. Choi, D. Lee, S. H. Noh, and O. Mutlu, "Amnesic cache management for non-volatile memory," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–13.

[60] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance trade-offs in using NVRAM write buffer for flash memory-based storage devices," *IEEE Trans. Comput.*, vol. 58, no. 6, pp. 744–758, Jun. 2009.

[61] Y. Kang, R. Pitchumani, T. Marlette, and E. L. Miller, "Muninn: A versioning flash key-value store using an object-based storage model," in *Proc. Int. Conf. Syst. Storage*, 2014, pp. 1–11.

[62] S. Kannan, A. Gavrilovska, and K. Schwan, "pVM: persistent virtual memory for efficient capacity scaling and object storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, p. 13.

[63] S. Kannan, A. Gavrilovska, K. Schwan, D. Milojicic, and V. Talwar, "Using active NVRAM for I/O staging," in *Proc. 2nd Int. Workshop Petascal Data Anal., Challenges Opportunities*, 2011, pp. 15–22.

[64] S. Keele, "Guidelines for performing systematic literature reviews in software engineering," Ver. 2.3, EBSE Tech. Rep., 2007.

[65] M. G. Khatib, P. H. Hartel, and H. W. Van Dijk, "Energy-efficient streaming using non-volatile memory," *J. Signal Process. Syst.*, vol. 60, no. 2, pp. 149–168, 2010.

[66] D. Kim and S. Kang, "Dual region write buffering: Making large-scale nonvolatile buffer using small capacitor in SSD," in *Proc. ACM Symp. Appl. Comput.*, 2015, pp. 2039–2046.

[67] H. Kim, "In-memory file system for non-volatile memory," in *Proc. Res. Adapt. Convergent Syst. Conf.*, 2013, pp. 479–484.

[68] J. Kim, C. Min, and Y. I. Eom, "Reducing excessive journaling overhead with small-sized NVRAM for mobile devices," *IEEE Trans. Consum. Electron.*, vol. 60, no. 2, pp. 217–224, May 2014.

[69] Y. Kim, A. Gupta, and B. Urgaonkar, "A temporal locality-aware page-mapped flash translation layer," *J. Comput. Sci. Technol.*, vol. 28, no. 6, pp. 1025–1044, 2013.

[70] A. Kolli *et al.*, "Delegated persist ordering," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2016, pp. 1–13.

[71] M. P. Komalan *et al.*, "Design exploration of a NVM based hybrid instruction memory organization for embedded platforms," *Design Autom. Embedded Syst.*, vol. 17, nos. 3–4, pp. 459–483, 2014.

[72] M. H. Kryder and C. S. Kim, "After hard drives—What comes next?" *IEEE Trans. Magn.*, vol. 45, no. 10, pp. 3406–3413, 2009.

[73] E. Lee, H. Bahn, and S. H. Noh, "A unified buffer cache architecture that subsumes journaling functionality via nonvolatile memory," *ACM Trans. Storage*, vol. 10, no. 1, pp. 1–17, 2014.

[74] E. Lee, H. Kang, H. Bahn, and K. G. Shin, "Eliminating periodic flush overhead of file I/O with non-volatile buffer cache," *IEEE Trans. Comput.*, vol. 65, no. 4, pp. 1145–1157, Apr. 2016.

[75] E. Lee, S. Yoo, J. Jang, and H. Bahn, "WIPS: A write-in-place snapshot file system for storage-class memory," *Electron. Lett.*, vol. 48, no. 17, pp. 16–17, Aug. 2012.

[76] H. G. Lee, "High-performance NAND and PRAM hybrid storage design for consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 56, no. 1, pp. 112–118, Feb. 2010.

[77] K. Lee, S. Ryu, and H. Han, "Performance implications of cache flushes for non-volatile memory file systems," in *Proc. ACM Symp. Appl. Comput.*, 2015, pp. 2069–2071.

[78] M. Lee, D. H. Kang, J. Kim, and Y. I. Eom, "M-clock: Migration-optimized page replacement algorithm for hybrid dram and pcm memory architecture," in *Proc. ACM Symp. Appl. Comput.*, 2015, pp. 2001–2006.

[79] S. Lee, J. Kim, M. Lee, H. Lee, and Y. I. Eom, "Last block logging mechanism for improving performance and lifetime on SCM-based file system," in *Proc. 8th Int. Conf. Ubiquitous Inf. Manage. Commun.*, 2014, pp. 38:1–38:4.

[80] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.

[81] Y. Lee, S. Jung, M. Choi, and Y. H. Song, "An efficient management scheme for updating redundant information in flash-based storage system," *Design Autom. Embedded Syst.*, vol. 14, no. 4, pp. 389–413, 2010.

[82] D. Li, X. Liao, H. Jin, Y. Tang, and G. Zhao, "Writeback throttling in a virtualized system with SCM," *Frontiers Comput. Sci.*, vol. 10, no. 1, pp. 82–95, 2016.

[83] G. Li, P. Zhao, L. Yuan, and S. Gao, "Efficient implementation of a multi-dimensional index structure over flash memory storage systems," *J. Supercomput.*, vol. 64, no. 3, pp. 1055–1074, 2011.

[84] H. Li and Y. Chen, "An overview of non-volatile memory technology and the implication for tools and architectures," in *Proc. Design, Autom. Test Eur. Conf. Exhib.*, 2009, pp. 731–736.

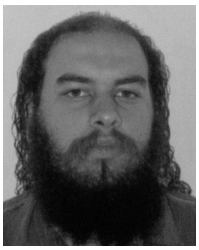
[85] H.-Y. Li, N.-X. Xiong, P. Huang, and C. Gui, "PASS: A simple, efficient parallelism-aware solid state drive I/O scheduler," *J. Zhejiang Univ. Sci. C*, vol. 15, no. 5, pp. 321–336, 2014.

[86] J. Li, Q. Zhuge, D. Liu, H. Luo, and E. H.-M. Sha, "A content-aware writing mechanism for reducing energy on non-volatile memory based embedded storage systems," *Design Autom. Embedded Syst.*, vol. 17, nos. 3–4, pp. 711–737, 2014.

[87] X. Li and K. Lu, "FCKPT:A fine-grained incremental checkpoint for PCM," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, 2011, pp. 2019–2023.

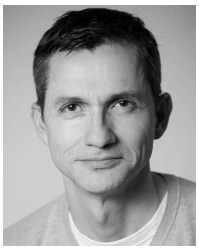
[88] Y. Li, Y. Wang, A. Jiang, and J. Bruck, "Content-assisted file decoding for nonvolatile memories," in *Proc. Conf. Rec. Asilomar Conf. Signals, Syst. Comput.*, 2012, pp. 937–941.
[89] S.-H. Lim and M.-K. Seo, "Efficient non-linear multimedia editing for non-volatile mobile storage," in *Proc. ACM Workshop Mobile Cloud Media Comput.*, 2010, pp. 59–64.
[90] R.-S. Liu, D.-Y. Shen, C.-L. Yang, S.-C. Yu, and C.-Y. M. Wang, "NVM duet: Unified working memory and persistent store architecture," *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 1, pp. 455–470, 2014.
[91] Y. Lu, J. Shu, and P. Zhu, "TxCache: Transactional cache using byte-addressable non-volatile memories in SSDs," in *Proc. Non-Volatile Memory Syst. Appl. Symp.*, 2014, pp. 1–6.
[92] Y. Lu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2009, pp. 257–270.
[93] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng, "Overview of emerging nonvolatile memory technologies," *Nanosc. Res. Lett.*, vol. 9, no. 1, p. 526, 2014.
[94] S. Mittal and J. S. Vetter, "A survey of software techniques for using non-volatile memories for storage and main memory systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1537–1550, May 2016.
[95] S. Mittal and J. S. Vetter, "EqualChance: Addressing intra-set write variation to increase lifetime of non-volatile caches," in *Proc. 2nd Workshop Interact. NVM/Flash Oper. Syst. Workloads*, 2014, pp. 1–10.
[96] (2016). *NVM Library*. [Online]. Available: http://github.com/pmem/nvml/
[97] S. Oikawa, "Independent kernel/process checkpointing on non-volatile main memory for quick kernel rejuvenation," in *Proc. Archit. Comput. Syst. (ARCS)*, vol. 8350. Cham, Switzerland: Springer, 2014, pp. 233–244. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-04891-8_20
[98] S. Oikawa, "Integration methods of main memory and file system management for non-volatile main memory and implications of file system structures," in *Proc. IEEE 16th Int. Symp. Object/Component/Service-Oriented Real-Time Distrib. Comput.*, Jun. 2013, pp. 1–8.
[99] S. Oikawa, "Non-volatile main memory management methods based on a file system," *SpringerPlus*, vol. 2014, p. 494, Sep. 2014.
[100] S. Oikawa and S. Miki, "File-based memory management for non-volatile main memory," in *Proc. 37th IEEE Annu. Comput. Softw. Appl. Conf.*, Jul. 2013, pp. 559–568.
[101] P. Olivier, J. Boukhobza, and E. Senn, "Micro-benchmarking flash memory file-system wear leveling and garbage collection: A focus on initial state impact," in *Proc. 15th IEEE Int. Conf. Comput. Sci. Eng.*, Dec. 2012, pp. 437–444.
[102] P. Olivier, J. Boukhobza, and E. Senn, "On benchmarking embedded Linux flash file systems," *ACM SIGBED Rev.*, vol. 9, no. 2, pp. 43–47, 2012.
[103] S. Park, T. Kelly, and K. Shen, "Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data," in *Proc. 8th ACM Eur. Conf. Comput. Syst.*, 2013, pp. 225–238.
[104] Y. Park and K. H. Park, "High-performance scalable flash file system using virtual metadata storage with phase-change RAM," *IEEE Trans. Comput.*, vol. 60, no. 3, pp. 321–334, Mar. 2011.
[105] Y. Park, S.-H. Lim, C. Lee, and K. H. Park, "PFFS: A scalable flash memory file system for the hybrid architecture of phase-change RAM and NAND flash," in *Proc. ACM Symp. Appl. Comput.*, 2008, pp. 1498–1503.
[106] T. Perez, N. L. V. Calazans, and C. A. F. De Rose, "System-level impacts of persistent main memory using a search engine," *Microelectron. J.*, vol. 45, no. 2, pp. 211–216, 2014.
[107] K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, "Systematic mapping studies in software engineering," in *Proc. 2th Int. Conf. Eval. Assessment Softw. Eng.*, 2007, pp. 68–77.
[108] Adrian Proctor. (2014). *Storage in the DIMM Socket*. [Online]. Available: http://www.snia.org/forums/sssi/NVDIMM
[109] S. Qiu and A. L. N. Reddy, "Exploiting superpages in a nonvolatile memory file system," in *Proc. IEEE Symp. Mass Storage Syst. Technol.*, Apr. 2012, pp. 1–5.
[110] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," *ACM SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 24–33, 2009.
[111] A. S. Ramasamy and P. Karantharaj, "File system and storage array design challenges for flash memory," in *Proc. Int. Conf. Green Comput. Commun. Elect. Eng.*, 2014, pp. 1–5.
[112] L. Ramos and R. Bianchini, "Exploiting phase-change memory in cooperative caches," in *Proc. Symp. Comput. Archit. High Perform. Comput.*, 2012, pp. 227–234.
[113] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM J. Res. Develop.*, vol. 52, nos. 4–5, pp. 465–479, 2008.
[114] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. 48th Int. Symp. Microarchitecture*, 2015, pp. 672–685.
[115] S. Ryu, K. Lee, and H. Han, "In-memory write-ahead logging for mobile smart devices with NVRAM," *IEEE Trans. Consum. Electron.*, vol. 61, no. 1, pp. 39–46, Feb. 2015.
[116] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid dram-NVM memory architecture," in *Proc. Conf. Design, Autom., Test Eur.*, 2016, pp. 936–941.
[117] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti, "An empirical study of file systems on NVM," in *Proc. 31st Symp. Mass Storage Syst. Technol.*, 2015, pp. 1–14.
[118] D. Seo and D. Shin, "WAM: Wear wear-out-aware memory management for SCRAM-based low power mobile systems," *IEEE Trans. Consum. Electron.*, vol. 59, no. 4, pp. 803–810, Nov. 2013.
[119] E. Sha, X. Chen, Q. Zhuge, L. Shi, and W. Jiang, "A new design of in-memory file system based on file virtual address framework," *IEEE Trans. Comput.*, vol. 65, no. 10, pp. 2959–2972, Oct. 2016.
[120] E. H.-M. Sha, Y. Jia, X. Chen, Q. Zhuge, W. Jiang, and J. Qin, "The design and implementation of an efficient user-space in-memory file system," in *Proc. 5th Non-Volatile Memory Syst. Appl. Symp.*, 2016, pp. 1–6.
[121] P. Snyder, "tmpfs: A virtual memory file system," in *Proc. Autumn EUUG Conf.*, 1990, pp. 241–248.
[122] Y. Son, H. Kang, H. Han, and H. Y. Yeom, "An empirical evaluation and analysis of the performance of NVM express solid state drive," *Cluster Comput.*, vol. 19, no. 3, pp. 1541–1553, 2016.
[123] Y. Son, N. Y. Song, H. Han, H. Eom, and H. Y. Yeom, "Design and evaluation of a user-level file system for fast storage devices," *Cluster Comput.*, vol. 18, no. 3, pp. 1075–1086, 2015.
[124] G. Sun *et al.*, "A Hybrid solid-state storage architecture for the performance, energy consumption, and lifetime improvement," in *Proc. 16th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Jan. 2010, pp. 1–12.
[125] J. Sun, X. Long, H. Wan, and J. Yang, "A checkpointing and instant-on mechanism for a embedded system based on non-volatile memories," in *Proc. IT Appl. Conf. Comput., Commun.*, 2014, pp. 173–178.
[126] D. Tuteja, E. L. Miller, S. A. Brandt, and N. K. Edel, "MRAMFS: A compressing file system for non-volatile RAM," in *Proc. 12th IEEE Annu. Int. Symp. Modeling Anal., Simulation Comput. Telecommun. Syst. (MASCOTS)*, Oct. 2011, pp. 596–603.
[127] R. Verma *et al.*, "Failure-atomic updates of application data in a Linux file system," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 203–211.
[128] D. Verneer, "Execute-in-place," *Memory Card Mag.*, Apr. 1991.
[129] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *Archit. Support Program. Lang. Oper. Syst.*, vol. 47, no. 4, pp. 91–104, 2011.
[130] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 14:1–14:14.
[131] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning, "Conquest: Better performance through a disk/persistent-RAM hybrid file system," in *Proc. USENIX Annu. Tech. Conf., Gen. Track*, 2002, pp. 15–28.
[132] C. Wang and S. Baskiyar, "Extending flash lifetime in secondary storage," *Microprocess. Microsyst.*, vol. 39, no. 3, pp. 167–180, 2015.
[133] C. Wu, G. Zhang, and K. Li, "Rethinking computer architectures and software systems for phase-change memory," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 12, no. 4, p. 33, 2016.
[134] C. H. Wu, W. Y. Chang, and Z. W. Hong, "A reliable non-volatile memory system: Exploiting file-system characteristics," in *Proc. 15th IEEE Pacific Rim Int. Symp. Dependable Comput. (PRDC)*, Nov. 2009, pp. 202–207.
[135] C. H. Wu, P. H. Wu, K. L. Chen, W. Y. Chang, and K. C. Lai, "A hotness filter of files for reliable non-volatile memory systems," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 4, pp. 375–386, Jul. 2015.
[136] X. Wu and A. L. N. Reddy, "SCMFS: A file system for storage class memory," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, pp. 1–11.

[137] F. Xia, D.-J. Jiang, J. Xiong, and N.-H. Sun, "A survey of phase change memory systems," *J. Comput. Sci. Technol.*, vol. 30, no. 1, pp. 121–144, 2015.
[138] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 323–338.
[139] J. J. Yang, M. D. Pickett, X. Li, D. A. A. Ohlberg, D. R. Stewart, and R. S. Williams, "Memristive switching mechanism for metal/oxide/metal nanodevices," *Nature Nanotechnol.*, vol. 3, no. 7, pp. 429–433, 2008.
[140] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 1–7.
[141] S.-K. Yoon, M.-Y. Bian, and S.-D. Kim, "An integrated memory-disk system with buffering adapter and non-volatile memory," *Des. Autom. Embedded Syst.*, vol. 17, nos. 3–4, pp. 609–626, 2013.
[142] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A reliable and highly-available non-volatile memory system," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 3–18, 2015.
[143] Z. Zhang, L. Ju, and Z. Jia, "Unified dram and NVM hybrid buffer cache architecture for reducing journaling overhead," in *Proc. Conf. Design, Autom., Test Eur.*, 2016, pp. 942–947.
[144] J. Zhao, S. Li, D. Yoon, Y. Xie, and N. Jouppi, "Kiln: closing the performance gap between systems with and without persistence support," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2013, pp. 421–432.
[145] R. Zwisler. (2014). *Add Persistent Memory Driver*. [Online]. Available: http://lwn.net/Articles/609755/



GIANLUCCA O. PUGLIA received the B.Sc. degree in computer science and the M.Sc. degree in operating systems and parallel processing from the Pontifical University of Rio Grande do Sul (PUCRS), in 2013 and 2017, respectively. His M.Sc. dissertation was on file system design for non-volatile memories. From 2015 to 2017, he was a Researcher of Hewlett-Packard research projects with focus on non-modern operating system designs and volatile memory storage, as part of his scholarship. His main areas of research interests include parallel and distributed systems, non-volatile memory, operating systems, and storage.

AVELINO FRANCISCO ZORZO received the Ph.D. degree in computer science from Newcastle University, U.K., in 1999. In 2012, he joined the Cybercrime and Computer Security Centre, Newcastle University, where he held a Postdoctoral position. He was the Dean of the Faculty of Informatics, Pontifical Catholic University of Rio Grande do Sul, Brazil, where he is currently a full-time Professor. His main research topics include security, fault tolerance, software testing, and operating systems. He was the Education Director of the Brazilian Computer Society. He is also a Coordinator of professional postgraduate accreditation with the Ministry of Education, Brazil.



CÉSAR A. F. DE ROSE received the B.Sc. degree in computer science from the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil, in 1990, the M.Sc. degree in computer science from PGCC, Federal University of Rio Grande do Sul, Porto Alegre, in 1993, and the Doctoral degree from the Karlsruhe Institute of technology, Karlsruhe, Germany, in 1998. In 1998, he joined the Faculty of Informatics, PUCRS, as an Associate Professor and a member of the Resource Management and Virtualization Group. In 2009, he founded the High Performance Computing Laboratory, PUCRS, where is currently a Senior Researcher. Since 2012, he has been a Full Professor with PUCRS. His research interests include resource management, dynamic provisioning and allocation, monitoring techniques (resource and application), application modeling, scheduling and optimization in parallel and distributed environments (cluster, grid, and cloud), and virtualization.

TACIANO D. PEREZ received the M.Sc. and Ph.D. degrees in computer science from the Pontifical Catholic University of Rio Grande do Sul, Brazil, in 2017. He is currently a Research and Development Software Engineer with ASML, Eindhoven, The Netherlands. He is with HP Labs, Porto Alegre, Brazil, leading research programs on non-volatile memory systems. His areas of expertise are non-volatile memory, persistence support for programming languages, and operating systems.

DEJAN MILOJICIC (F'10) received the B.Sc. and M.Sc. degrees from Belgrade University, Serbia, and the Ph.D. degree from Kaiserslautern University, Germany. He was with the OSF Research Institute, Cambridge, MA, and also with the Mihajlo Pupin Institute, Belgrade, Serbia. He was the Technical Director of Open Cirrus Cloud Computing Testbed, with academic, industrial, and government sites in USA, Europe, and Asia. He taught cloud management course with SJSU. Since 1998, he has been a Distinguished Technologist with Hewlett Packard Labs, Palo Alto, CA, leading system software teams over four continents and projects with budgets of millions of U.S. dollars. He has published two books and 180 papers. He holds 31 granted patents. He was an ACM Distinguished Engineer, in 2008. He was on eight Ph.D. dissertation committees. As the President of the IEEE Computer Society, in 2014, he started Tech Trends, the top viewed CS news. As the Industry Engagement Chair, he started the IEEE Infrastructure'18 Conference.